Everything You Have Always Wanted to Know about the Playstation

But Were Afraid to Ask.

Version 1.1

Compiled \ Edited By
Joshua Walker

Table of Contents

1. Introduction
2. History
3. The R3000A

- Overview
- The R3000A instruction set
- R3000A opcode encoding

1. Memory

- Memory Map
- Virtual Memory

- The System Control Coprocessor (Cop0)

- Exception Handling
- Breakpoint management
- DMA

1. Video

- Overview

- The Graphics Processing Unit (GPU)
- The Graphics Transformation Engine (GTE)
- The Motion Decoder (MDEC)

1. Sound

- The Sound Processing Unit (SPU)

1. CD-ROM
2. Root Counters
3. Controllers
4. Memory Cards
5. Serial port I/O
6. Parallel port I/O

Appendices

1. Number Systems
2. BIOS functions
3. GPU command listing
4. Glossary of terms
5. Works cited – Bibliography

Introduction

This project to document the Playstation stated about a year ago. It started with the utter disgust I had for Sony of America after suing Bleem over the PSX emulation technology. I saw the ugliness of a huge multinational company try to destroy two guys who had a good idea and even tried to *share* it with them. It made me sick. I wanted to do something to help, but alas I had no money, (I still don't) but I did buy a Bleem CD to support them.

I decided to start this little project. Partially to prove to Sony, but mostly to prove to myself, that coming up with the data to create you own emulator was not that hard. I also wanted to show that behind that gray box that so many people hold dear. It's just a computer with no keyboard, that plugs into your TV. It's one thing to think that you were spending $250 on a new PSX, but it's another to realize that the CPU costs $5.99 from LSI.

Kind of puts thing into perspective, doesn't it.

I'm not a programmer. I've never worked for sony, and I never signed a Non-Disclosure Agreement with them. I just took my PSX apart, found out what made it tick, and put it back together. I also scoured the web looking for material that I could find. I never looked at any of Sony's official documentation and never took any thing you would have to have a license to get. Such as PSY-Q. I mostly poked at emulators to see how they worked. Bleem was only 512k at the time and was pretty easy to see how it functioned without even running it through a dissembler. PSEmu had an awesome debugger so I can see how a PSX ran even without caelta.

I want this documentation to be freely available. Anyone can use it. From the seasoned PSX programmer to the lurking programmer read to make the next big emulator. If there is a discrepancy in my doc, please fix it. Tear out parts that are wrong and correct it so it's better that what I have now. I wanted to shoot for a 75% accuracy rating. I think I got it, but I don't know. Most of the stuff in here is hearsay and logical deductions. Much of it is merely a guess.

Of course there is the standard disclaimer, all trademarks are of the appropriate owners and that this documentation is not endorsed by Sony or Bleem in any way. You are, once again, free to give this away, trade it, or do what you will. It's not mine anymore. It's everybody's. Do with it what you please. Oh, and if your PSX blows up or melts down due to this documentation, sorry. I can't assure the validity of *any* info other that I didn't get it from Sony's official documentation. I'm not responsible to what you do to your machine.

In closing I wish to apologize for the way this introduction was written as it's 2:00 in the morning. I have a wedding to get to at 10:30 and I've been up for the last three days finishing the darn thing. I wish to thank everyone who supported me. Janice, for believing in me and My girlfriend Kim who put it with the long nights in front of the computer writing and the long days in fornt of the Playstation claming I was "doing research" while playing FF8. I can't think of anything more to say. Have fun with this

-Joshua Walker

4/29/00

2:34am

84905

History

Prologue B.P. (Before PlayStation)

Before the release of the PlayStation, Sony had never held a large portion of the videogames market. It had made a few forays into the computer side of things, most notably in its involvement with the failed MSX chip in the early 80's, but it wasn't until the advent of CD-ROM technology that Sony could claim any market share. A joint venture with the Dutch company Philips had yielded the CD-ROM/XA, an extension of the CD-ROM format that combined compressed audio, and visual and computer data and allowed both to be accessed simultaneously with the aid of extra hardware. By the late 80's, CD-ROM technology was being assimilated, albeit slowly, into the home computer market, and Sony was right there along side it. But they wanted a bigger piece of the pie.

1988 Sony Enters The Arena

By 1988, the gaming world was firmly gripped in Nintendo's 8-bit fist. Sega had yet to make a proper showing, and Sony, although hungry for some action, hadn't made any moves of its own.

Yet.

Sony's first foray into the gaming market came in 1988, when it embarked on a deal with Nintendo to develop a CD-ROM drive for the Super NES, not scheduled to be released for another 18 months. This was Sony's chance to finally get involved in the home videogame market. What better way to enter that arena than on the coat-tails of the world's biggest gaming company?

Using the same Super Disc technology as the proposed SNES drive, Sony began development on what was to eventually become the PlayStaion. Initially called the Super Disc, it was supposed to be able to play both SNES cartridges and CD-ROMs, of which Sony was to be the "sole worldwide licenser," as stated in the contract. Nintendo was now to be at the mercy of Sony, who could manufacture their own CDs, play SNES carts, and play Sony CDs. Needless to say, Nintendo began to get worried.

1991 Multimedia Comes Home

1991 saw the commercial release of the multimedia machine in the form of Philips' CD-I, which had initially been developed jointly by both Philips and Sony until mounting conflicts resulted in a parting of ways. Multimedia, with the current rise of the CD-ROM, was seen as the next big thing. And although the CD-I was too expensive for the mass market, its arrival cemented the CD-ROM as a medium for entertainment beyond the computer.

June 1991 Treachery At The 11th Hour

I n June of 1991, at the Chicago CES (Consumer Electronics Show), Sony officially announced the Play Station (space intentional). The Play Station would have a port to play Super Nintendo cartridges, as well as a CD-ROM drive that would play Sony Super Discs. The machine would be able to play videogames as well as other forms of interactive entertainment, as was considered important at the time.

Sony intended to draw on its family of companies, including Sony Music and Columbia Pictures, to develop software. Olaf Olafsson, then chief of Sony Electronic Publishing, was seen on the set of Hook, Steven Spielberg's new Peter Pan movie, presumably deciding how the movie could be worked into a game for the fledgling Play Station. In Fortune magazine, Olafsson was quoted as saying "The video-game business...will be much more interesting (than when it was cartridge based). By owning a studio, we can get involved right from the beginning, during the writing of the movie."

By this point, Nintendo had had just about all it could take. On top of the deal signed in 1988, Sony had also contributed the main audio chip to the cartridge-based Super NES.

The Ken Kutaragi-designed chip was a key element to the system, but was designed in such a way as to make effective development possible only with Sony's expensive development tools. Sony had also retained all rights to the chip, which further exaserbated Nintendo.

The day after Sony announced its plans to begin work on the Play Station, Nintendo made an announcement of its own. Instead of confirming its alliance with Sony, as everyone expected, Nintendo announced it was working with Philips, Sony's longtime rivals, on the SNES CD-ROM drive. Sony was understandably furious.

Because of their contract-breaking actions, Nintendo not only faced legal repercussions from Sony, but could also experience a serious backlash from the Japanese business community. Nintendo had broken the unwritten law that a company shouldn't turn against a reigning Japanese company in favor of a foreign one.

However, Nintendo managed to escape without a penalty. Because of their mutual involvement, it would be in the best interests of both companies to maintain friendly relations. Sony, after all, was planning a port for SNES carts, and Nintendo was still using the Sony audio chip.

1992 The Smoke Clears

By the end of 1992, most of the storm had blown over. Despite a deal penned between Sega, one of Nintendo's biggest competitors, and Sony, whereby Sony would produce software for the proposed Sega Multimedia Entertainment System, negotiations were reached with Nintendo. In October of 1992, it was announced that the two companies' CD-ROM players would be compatible. The software licensing for the proposed 32-bit machines was awarded to Nintendo, with Sony receiving only minimal licensing royalties. Nintendo had won this battle, but hadn't won the war. Not by a long shot.

The first Play Station never made it out of the factories. Apparently, about 200 were produced, and some software even made it to development. For whatever reason, whether it was because of the tough licensing deal with Nintendo, or the predicted passing of masked ROM (cartridge-based) technology, Sony scrapped its prototype. Steve Race, Sony Computer Entertainment Of America's (SCEA) then CEO, stated, "Since the deal with Nintendo didn't come to fruition we decided to put games on a back burner and wait for the next category. Generally, the gaming industry has a seven-year product life-cycle, so we bided our time until we could get in on the next cycle."

1993 The Next Cycle

After returning to the drawing boards, Sony revealed the PS-X, or PlayStation-X. Gone was the original cartridge port, as were the plans for multimedia. Apparently, Sony had visited 3DO when Trip Hawkins was selling his hardware and had come away unimpressed, saying it was "nothing new." The PS-X was to be a dedicated game-machine, pure and simple. Steve Race said in Next Generation magazine, "We designed the PlayStation to be the best game player we could possibly make. Games really are multimedia, no matter what we want to call it. The conclusion is that the PlayStation is a multimedia machine that is positioned as the ultimate game player."

Key to Sony's battle plan was the implementation of 3D into its graphics capabilities, a move that many felt was critical to any kind of future success. At the core of the PlayStation's 3D prowess was the R3000 processor, operating at 33 Mhz and 30 MIPS (millions of instructions per second). While this may seem fairly

average for a RISC CPU, it's the PlayStation's supplementary custom hardware, designed by Ken Kutaragi (who had previously designed the key audio chip for the SNES), that provides the real power. The CPU relies heavily on Kutaragi's VLSI (very large scale integration) chip to provide the speed necessary to process complex 3D graphics quickly.

The CPU is backed up by the GPU (Graphics Processing Unit), which takes care of all the data from the CPU and passes the results to the 1024K of dual-ported VRAM, which stores the current frame buffer and allows the picture to be displayed on-screen. Part of this picture involves adding special effects such as transparency and fog, something that the PlayStation excels at. This was to be the most impressive display of hardware the home gaming world had ever seen

1994 Third Party Round Up

There was no doubt that Sony could deliver the hardware. After all, Sony was one of the world's largest manufacturers of electronics. There was no denying though, that Sony was extremely green when it came to videogames. And no one knew it better than Sony.

Not wanting to end up like Atari or 3DO, Sony set about rounding up third party developers, assembling an impressive 250 developing parties in Japan alone. Sony also knew it had to gain the support of the millions of arcade-going gamers if it was to succeed. The involvement of Namco, Konami, and Williams helped ensure Sony would be able to compete with the arcade-savvy Sega on its own turf. Namco's Ridge Racer was the natural choice to be the flagship launch game, and Williams' Mortal Kombat III, previously promised to Nintendo for their Ultra 64, could be tested in the arcades using the new PS-X board.

Perhaps Sony's most controversial acquisition was the purchase of Psygnosis, a relatively unknown European developer, for $48 million. Sony needed a strong in-house development team, and Psygnosis' Lemmings seemed to point at good things. While the purchase confused many at the time, prompting vocal outcries from naysayers and competitors alike, Psygnosis has since proven them all wrong. Sony Interactive Entertainment, as Psygnosis was renamed, has been responsible for some of the PlayStation's best games, including WipeOut and Destruction Derby.

Sony's acquisition of Psygnosis yielded another fruit as well: the development system. SN Systems, co-owned by Andy Beveridge and Martin Day, had been publishing its development software through Psygnosis under the PSY-Q moniker. Sony originally had been planning on using expensive, Japanese MIPS R4000-based machines that would be connected to the prototype PS-X box. Having become accustomed to developing on the PC, Psygnosis gave Beveridge and Day first crack at creating a PlayStation development system that would work with a standard PC.

The two men worked through Christmas and New Year's, around the clock, eventually completing the GNU-C compiler and the source-level debugger. Psygnosis quickly arranged a meeting for SN and Sony at the Winter CES in Las Vegas, 1994. Fortunately, Sony liked the PSY-Q alternative and decided to work with SN Systems on cendensing the software onto two PC-compatible cards. Thus,

an afordable and, more importantly, universally compatible PlayStation development station was born.

December 3, 1994 We Have Lift Off

On December 3, 1994, the PlayStation was finally released in Japan, one week after the Sega Saturn. The initial retail cost was 37,000 yen, or about $387. Software available at launch included King's Field, Crime Crackers, and Namco's Ridge Racer, the PlayStation's first certifiable killer app. It was met with long lines across Japan, and was hailed by Sony as their most important product since the WalkMan in the late 1970's.

Also available at launch were a host of peripherals, including: a memory card to save high scores and games; a link cable, whereby you could connect two PlayStations and two TVs and play against a friend; a mouse with pad for PC ports; an RFU Adaptor; an S-Video Adaptor; and a Multitap Unit. Third party peripherals were also available, including Namco's Negcon.

The look of the PlayStation was dramatically different than the Saturn, which was beige (in Japan), bulky, and somewhat clumsy looking. In contrast, the PlayStation was slim, sleek, and gray, with a revolutionary controller that was years ahead of the Saturn's SNES-like pad. The new PSX joypad provided unheardof control by adding two more buttons on the shoulder, making a total of eight buttons. The two extended grips also added a new element of control. Ken Kutaragi realized the importance of control when dealing with 3 Dimensional game worlds. "We probably spent as much time on the joypad's development as the body of the machine. Sony's boss showed special interest in achieving the final version so it has his seal of approval." To Sony's delight, the PlayStation sold more than 300,000 units in the first 30 days. The Saturn claimed to have sold 400,000, but research has shown that number to be misleading. The PSX sold through (to customers) 97% of its stock, while many Saturns were still sitting on the shelves. These misleading numbers were to be quoted by Sega on many occasions, and continued even after the US launch.

1995 Setting Up House

By mid-1995, Sony had set its sights firmly on the United States. Sony Computer Entertainment of America was created and housed in Foster City, California, in the heart of Silicon Valley. Steve Race, formerly of Atari, was appointed as president and CEO of the new branch of Sony. The accumulation of third party developers continued apace, with over 100 licenses in the US and 270 licenses in Japan secured. Steve Race said, "We've allowed people to come in and to play on the PlayStation - and at a much more reasonable cost than has been done in the old days with Nintendo and Sega." Sony's development strategy had paid off, with over 700 development units having been shipped out worldwide.

May 11, 1995 Victory At E3

The Electronic Entertainment Expo (E3) was held in Los Angeles from May 11 to 13, 1995, and was the United State's first real look at the PlayStation. Sony made a huge impression at the show with their (rumored) $4 million booth and surprise appearance by Michael Jackson. The PSX was definitely the highlight of the show, besting Sega's Saturn and Nintendo's laughable Virtual Boy.

The launch software was also displayed, with WipeOut and Namco's Tekken and Ridge Racer drawing the most praise. Sony also announced the unit would not be bundled with Ridge Racer, as was previously assumed.

Overall, Sony made a very formidable showing at E3. They had already proven themselves in Japan and were close on Sega's heels. Over the course of the next year they would overtake Sega and conquer Japan as their own. Now they were poised to do the same in America.

September 9, 1995 You Are Not Ready

The PlayStation launched in the United States on September 9, 1995 to instant success. Although it retailed for $299, that was still $100 less than the Sega Saturn. Over 100,000 units were already presold at launch, and 17 games were available. Stores reported sell-outs across the country, and sold out of many games and peripherals as well, including second controllers and memory cards.

Sony's initial marketing strategy seemed to be aimed at an older audience than the traditional 8-16 year old demographic of the past. With the tag line "U R Not E" (the "E" being red) and a rumored $40 million to spend on launch marketing, Sony swiftly positioned itself as the market leader. To further cement their audience demographic, Sony sponsored the 1995 MTV Music Awards.

Epilogue What A Year

By the US launch, Sony had sold over one million PlayStations in Japan alone. Since the US launch, as of late 1996, the PlayStation has sold over 7 million units worldwide, with close to two million of those being in the US alone. In May of 1996, Sony dropped the price of the PlayStation to $199, making it even more attractive to buy.

Like Japan, America and Europe embraced the PlayStation as their next-gen console of choice. The demographic of PlayStation owners has fallen in years steadily from twenty-somethings to the younger age bracket so coveted by Nintendo. In fact, many former Nintendo loyalists, tired of waiting for the Nintendo 64 to be released, bought PlayStations and are now happier for it. With close to 200 games available by Christmas 1996, it's easy to see why. This really is the ultimate gaming console!

The R3000A

Overview

The heart of the PSX is a slightly modified R3000A CPU from MIPS and LSI. This is a 32 bit Reduced Instruction Set Controller (RISC) processor that clocks at 33.8688 MHz. It has an operating performance of 30 million instructions per second. In addition, it has an Internal instruction cache of 4 KB, a data cache of 1 KB and has a bus transfer rate of 132 MB/sec. It has internally one Arithmetic/Logic unit (ALU), One shifter, and totally lacks an FPU or floating point unit. The R3000A is configured for litle-endian byte order and defines a word as 32-bits, a half-word, as 16-bits, and a byte as 8-bits.

The PSX has two coprocessors, cop0, the System Control coprocessor, and cop2, the GPU or Graphics Processing Unit. These are covered later on in this document.

Instruction cache

The PSX's R3000A contains 4 KB of instruction cache. The instruction cache is organized with a line size of 16 bytes. This should achieve hit rate of around 80%. The cache is implemented using physical address and tags, as opposed to virtual ones.

Data cache

The PSX's R3000A incorporates an on-chip data cache of 1KB, organized as a line size of 4 bytes (one word). This also should achieve hit rates of 80% in most applications. This also is a directly mapped physical address cache. The data cache is implemented as a write through cache, to maintain that the main memory is the same as the internal cache. In order to minimize processor stalls due to data write operations, the bus interface unit uses a 4–deep write buffer which captures address and data at the processor execution rate, allowing it to be retired to main memory at a much slower rate without impacting system performance.

32 bit architecture

The R3000A uses thirty-two 32-bit registers, a 32 bit program counter, and two 32 bit registers for multiply/divide functions. The following table lists the registers by register number, name, and usage.

General Purpose Registers

Register number	Name	Usage
R0	ZR	Constant Zero
R1	AT	Reserved for the assembler
R2-R3	V0-V1	Values for results and expression evaluation
R4-R7	A0-A3	Arguments
R8-R15	T0-T7	Temporaries (not preserved across call)
R16-R23	S0-S7	Saved (preserved across call)
R24-R25	T8-T9	More temporaries (not preserved across call)
R26-R27	K0-K1	Reserved for OS Kernel
R28	GP	Global Pointer
R29	SP	Stack Pointer
R30	FP	Frame Pointer
R31	RA	Return address (set by function call)

Multiply/Divide result Registers and Program counter

Name	Description
HI	Multiplication 64 bit high result or division remainder
LO	Multiplication 64 bit low result or division quotient
PC	Program Counter

Even though all general purpose registers have different names, they are all treated the same except for two. The R0 (ZR) register is hardwired as zero. The Second exception is R31 (RA) which is used at a link register when link or jump routines are called. These instructions are used in subroutine calls, and the subroutine return address is placed in register R31. This register can be written to or read as a normal register in other operations.

R3000A Instruction set

The instruction encoding is based on the MIPS architecture. The means that there are three types of instruction encoding.

I-Type (Immediate)

op	rs	rt	immediate

J-Type (Jump)

op	target

R-Type (Register)

op	rs	rt	rd	shamt	funct

where:

op	**is a 6-bit operation code**
rs	**is a five bit source register specifier**
rt	**is a 5-bit target register or branch condition**
immediate	**is a 16-bit immediate, or branch or address displacement**
target	**is a 26-bit jump target address**
rd	**is a 5-bit destination register specifier**
shamt	**is a 5-bit shift amount**
funct	**is a 6-bit function field**

The R3000A instruction set can be divided into the following basic groups:

Load/Store instructions move data between memory and the general registers. They are all encoded as “I-Type” instructions, and the only addressing mode implemented is base register plus signed, immediate offset. This directly enables the use of three distinct addressing modes: register plus offset; register direct; and immediate.

Computational instructions perform arithmetic, logical, and shift operations on values in registers. They are encoded as either “R-Type” instructions, when both source operands as well as the result are general registers, and “I-Type”, when one of the source operands is a 16-bit immediate value. Computational instructions use a three address format, so that operations don’t needlessly interfere with the contents of source registers.

Jump and Branch instructions change the control flow of a program. A Jump instruction can be encoded as a “J-Type” instruction, in which case the Jump target address is a paged absolute address formed by combining the 26-bit immediate value with four bits of the Program Counter. This form is used for subroutine calls. Alternately, Jumps can be encoded using the “R-Type” format, in which case the target address is a 32-bit value contained in one of the general registers. This form is typically used for returns and dispatches. Branch operations are encoded as “I-Type” instructions. The target address is formed from a 16-bit displacement relative to the Program Counter. The Jump and Link instructions save a return address in Register r31. These are typically used as subroutine calls, where the subroutine return address is stored into r31 during the call operation.

Co-Processor instructions perform operations on the co-processor set. Co-Processor Loads and Stores are always encoded as "I-Type" instructions; co-processor operational instructions have co-processor dependent formats. In the R3000A, the System Control Co-Processor (cop0) contains registers which are used in memory management and exception handling.

Special instructions perform a variety of tasks, including movement of data between special and general registers, system calls, and breakpoint operations. They are always encoded as "R-Type" instructions.

INSTRUCTION SET SUMMARY

The following table describes The assembly instructions for the R3000A. Please refer to the appendix for more detail about opcode encoding

Load and Store Instructions

Instruction	Format and Description
Load Byte	**LB rt, offset (base)** **Sign-extend 16-bit offset and add to contents of register base to form address.** **Sign-extend contents of addressed byte and load into rt.**
Load Byte Unsigned	**LBU rt, offset (base)** **Sign-extend 16-bit offset and add to contents of register base to form address.** **Zero-extend contents of addressed byte and load into rt.**
Load Halfword	**LH rt, offset (base)** **Sign-extend 16-bit offset and add to contents of register base to form address.** **Sign-extend contents of addressed byte and load into rt.**
Load Halfword Unsigned	**LHU rt, offset (base)** **Sign-extend 16-bit offset and add to contents of register base to form address.** **Zero-extend contents of addressed byte and load into rt.**
Load Word	**LW rt, offset (base)** **Sign-extend 16-bit offset and add to contents of register base to form address.**

	Load contents of addressed word into register rt.
Load Word Left	**LWL rt, offset (base)** **Sign-extend 16-bit offset and add to contents of register base to form address.** **Shift addressed word left so that addressed byte is leftmost byte of a word.** **Merge bytes from memory with contents of register rt and load result into register rt.**
Load Word Right	**LWR rt, offset (base)** **Sign-extend 16-bit offset and add to contents of register base to form address.** **Shift addressed word right so that addressed byte is rightmost byte of a word.** **Merge bytes from memory with contents of register rt and load result into register rt.**
Store Byte	**SB rt, offset (base)** **Sign-extend 16-bit offset and add to contents of register base to form address.** **Store least significant byte of register rt at addressed location.**
Store Halfword	**SH rt, offset (base)** **Sign-extend 16-bit offset and add to contents of register base to form address.** **Store least significant halfword of register rt at addressed location.**
Store Word	**SW rt, offset (base)** **Sign-extend 16-bit offset and add to contents of register base to form address.** **Store least significant word of register rt at addressed location.**
Store Word Left	**SWL rt, offset (base)** **Sign-extend 16-bit offset and add to contents of register base to form address.** **Shift contents of register rt right so that leftmost byte of the word is in position of addressed byte. Store bytes containing original data into corresponding bytes at addressed byte.**
Store Word Right	**SWR rt, offset (base)** **Sign-extend 16-bit offset and add to contents of register base**

	to form address. Shift contents of register rt left so that rightmost byte of the word is in position of addressed byte. Store bytes containing original data into corresponding bytes at addressed byte.

Computational Instructions

ALU Immediate Operations

Instruction	Format and Description
ADD Immediate	ADDI rt, rs, immediate Add 16-bit sign-extended immediate to register rs and place 32-bit result in register rt . Trap on two's complement overflow.
ADD Immediate Unsigned	ADDIU rt, rs, immediate Add 16-bit sign-extended immediate to register rs and place 32-bit result in register rt . Do not trap on overflow.
Set on Less Than Immediate	SLTI rt, rs, immediate Compare 16-bit sign-extended immediate with register rs as signed 32-bit integers. Result = 1 if rs is less than immediate; otherwise result = 0. Place result in register rt.
Set on Less Than Unsigned Immediate	SLTIU rt, rs, immediate Compare 16-bit sign-extended immediate with register rs as unsigned 32-bit integers. Result = 1 if rs is less than immediate; otherwise result = 0. Place result in register rt. Do not trap on overflow.
AND Immediate	ANDI rt, rs, immediate Zero-extend 16-bit immediate, AND with contents of register rs and place result in register rt.
OR Immediate	ORI rt, rs, immediate Zero-extend 16-bit immediate, OR with contents of register rs and place result in register rt.
Exclusive OR Immediate	XORI rt, rs, immediate Zero-extend 16-bit immediate, exclusive OR with contents of register rs and place result in register rt.
Load Upper Immediate	LUI rt, immediate

	Shift 16-bit immediate left 16 bits. Set least significant 16 bits of word to zeroes. Store result in register rt.

Three Operand Register-Type Operations

Instruction	Format and Description
Add	**ADD rd, rs, rt** **Add contents of registers rs and rt and place 32-bit result in register rd. Trap on two's complement overflow.**
ADD Unsigned	**ADDU rd, rs, rt** **Add contents of registers rs and rt and place 32-bit result in register rd. Do not trap on overflow.**
Subtract	**SUB rd, rs, rt** **Subtract contents of registers rt and rs and place 32-bit result in register rd. Trap on two's complement overflow.**
Subtract Unsigned	**SUBU rd, rs, rt** **Subtract contents of registers rt and rs and place 32-bit result in register rd. Do not trap on overflow.**
Set on Less Than	**SLT rd, rs, rt** **Compare contents of register rt to register rs (as signed 32-bit integers).** **If register rs is less than rt, result = 1; otherwise, result = 0.**
Set on Less Than Unsigned	**SLTU rd, rs, rt** **Compare contents of register rt to register rs (as unsigned 32-bit integers). If register rs is less than rt, result = 1; otherwise, result = 0.**
AND	**AND rd, rs, rt** **Bit-wise AND contents of registers rs and rt and place result in register rd.**
OR	**OR rd, rs, rt** **Bit-wise OR contents of registers rs and rt and place result in register rd.**
Exclusive OR	**XOR rd, rs, rt**

	Bit-wise Exclusive OR contents of registers rs and rt and place result in register rd.
NOR	NOR rd, rs, rt Bit-wise NOR contents of registers rs and rt and place result in register rd.

Shift Operations

Instruction	Format and Description
Shift Left Logical	SLL rd, rt, shamt Shift contents of register rt left by shamt bits, inserting zeroes into low order bits. Place 32-bit result in register rd.
Shift Right Logical	SRL rd, rt, shamt Shift contents of register rt right by shamt bits, inserting zeroes into high order bits. Place 32-bit result in register rd.
Shift Right Arithmetic	SRA rd, rt, shamt Shift contents of register rt right by shamt bits, sign-extending the high order bits. Place 32-bit result in register rd.
Shift Left Logical Variable	SLLV rd, rt, rs Shift contents of register rt left. Low-order 5 bits of register rs specify number of bits to shift. Insert zeroes into low order bits of rt and place 32-bit result in register rd.
Shift Right Logical Variable	SRLV rd, rt, rs Shift contents of register rt right. Low-order 5 bits of register rs specify number of bits to shift. Insert zeroes into high order bits of rt and place 32-bit result in register rd.
Shift Right Arithmetic Variable	SRAV rd, rt, rs Shift contents of register rt right. Low-order 5 bits of register rs specify number of bits to shift. Sign-extend the high order bits of rt and place 32-bit result in register rd.

Multiply and Divide Operations

Instruction	Format and Description
Multiply	MULT rs, rt Multiply contents of registers rs and rt as twos complement values. Place 64-bit result in special registers HI/LO

Multiply Unsigned	**MULTU rs, rt** **Multiply contents of registers rs and rt as unsigned values. Place 64-bit result in special registers HI/LO**
Divide	**DIV rs, rt** **Divide contents of register rs by rt treating operands as twos complements values. Place 32-bit quotient in special register LO, and 32-bit remainder in HI.**
Divide Unsigned	**DIVU rs, rt** **Divide contents of register rs by rt treating operands as unsigned values. Place 32-bit quotient in special register LO, and 32-bit remainder in HI.**
Move From HI	**MFHI rd** **Move contents of special register HI to register rd.**
Move From LO	**MFLO rd** **Move contents of special register LO to register rd.**
Move To HI	**MTHI rd** **Move contents of special register rd to special register HI.**
Move To LO	**MTLO rd** **Move contents of register rd to special register LO.**

Jump and Branch Instructions

Jump Instructions

Instruction	Format and Description
Jump	**J target** **Shift 26-bit target address left two bits, combine with high-order4 bits of PC and jump to address with a one instruction delay.**
Jump and Link	**JAL target** **Shift 26-bit target address left two bits, combine with high-order 4 bits of PC and jump to address with a one instruction delay. Place address of instruction following delay slot in r31 (link register).**
Jump Register	**JR rs**

	Jump to address contained in register rs with a one instruction delay.
Jump and Link Register	JALR rs, rd Jump to address contained in register rs with a one instruction delay. Place address of instruction following delay slot in rd.

Branch Instructions

Instruction	Format and Description
	Branch Target: All Branch instruction target addresses are computed as follows: Add address of instruction in delay slot and the 16-bit offset (shifted left two bits and sign-extended to 32 bits). All branches occur with a delay of one instruction.
Branch on Equal	BEQ rs, rt, offset Branch to target address if register rs equal to rt
Branch on Not Equal	BNE rs, rt, offset Branch to target address if register rs not equal to rt.
Branch on Less than or Equal Zero	BLEZ rs, offset Branch to target address if register rs less than or equal to 0.
Branch on Greater Than Zero	BGTZ rs, offset Branch to target address if register rs greater than 0.
Branch on Less Than Zero	BLTZ rs, offset Branch to target address if register rs less than 0.
Branch on Greater than or Equal Zero	BGEZ rs, offset Branch to target address if register rs greater than or equal to 0.
Branch on Less Than Zero And Link	BLTZAL rs, offset Place address of instruction following delay slot in register r31 (link register). Branch to target address if register rs less than 0.
Branch on greater than or Equal Zero And Link	BGEZAL rs, offset Place address of instruction following delay slot in register r31 (link register). Branch to target address if register rs is greater than or equal to 0.

Special Instructions

Instruction	Format and Description
System Call	**SYSCALL** **Initiates system call trap, immediately transferring control to exception handler. More information on the PSX SYSCALL routines are covered later on.**
Breakpoint	**BREAK** **Initiates breakpoint trap, immediately transferring control to exception handler.**

More information on the PSX SYSCALL routines are covered later on.

Co-processor Instructions

Instruction	Format and Description
Load Word to Co-processor	**LWCz rt, offset (base)** **Sign-extend 16-bit offset and add to base to form address. Load contents of addressed word into co-processor register rt of co-processor unit z.**
Store Word from Co-processor	**SWCz rt, offset (base)** **Sign-extend 16-bit offset and add to base to form address. Store contents of co-processor register rt from co-processor unit z at addressed memory word.**
Move To Co-processor	**MTCz rt, rd** **Move contents of CPU register rt into co-processor register rd of co-processor unit z.**
Move from Co-processor	**MFCz rt,rd** **Move contents of co-processor register rd from co-processor unit z to CPU register rt.**
Move Control To Co-processor	**CTCz rt,rd**

	Move contents of CPU register rt into co-processor control register rd of co-processor unit z.
Move Control From Co-processor	CFCz rt,rd Move contents of control register rd of co-processor unit z into CPU register rt.
Move Control To Co-processor	COPz cofun Co-processor z performs an operation. The state of the R3000A is not modified by a co-processor operation.

System Control Co-processor (COP0) Instructions

Instruction	Format and Description
Move To CP0	MTC0 rt, rd Store contents of CPU register rt into register rd of CP0. This follows the convention of store operations.
Move From CP0	MFC0 rt, rd Load CPU register rt with contents of CP0 register rd.
Read Indexed TLB Entry	TLBR Load EntryHi and EntryLo registers with TLB entry pointed at by Index register.
Write Indexed TLB Entry	TLBWI Load TLB entry pointed at by Index register with contents of EntryHi and EntryLo registers.
Write Random TLB Entry	TLBWR Load TLB entry pointed at by Random register with contents of EntryHi and EntryLo registers.
Probe TLB for Matching Entry	TLBP Entry Load Index register with address of TLB entry whose contents match EntryHi and EntryLo. If no TLB entry matches, set high-order bit of Index register.

Restore From Exception	**RFE** **Restore previous interrupt mask and mode bits of status register into current status bits. Restore old status bits into previous status bits.**

R3000A OPCODE ENCODING

The following shows the opcode encoding for the MIPS architecture.

OPCODE

Bits	28...26							
38…29	0	1	2	3	4	5	6	7
0	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	COP3	†	†	†	†
3	†	†	†	†	†	†	†	†
4	LB	LH	LWL	LW	LBU	LHU	LWR	†
5	SB	SH	SWL	SW	†	†	SWR	†
6	LWC0	LWC1	LWC2	LWC3	†	†	†	†
7	SWC0	SWC1	SWC2	SWC3	†	†	†	†

SPECIAL

Bits	2…0							

5…3	0	1	2	3	4	5	6	7
0	SLL	†	SRL	SRA	SLLV	†	SRLV	SRAV
1	JR	JALR	†	†	SYSCALL	BREAK	†	†
2	MFHI	MTHI	MFLO	MTLO	†	†	†	†
3	MULT	MULTU	DIV	DIVU	†	†	†	†
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	†	†	SLT	SLTU	†	†	†	†
6	†	†	†	†	†	†	†	†
7	†	†	†	†	†	†	†	†

BCOND

Bits	8…16							
20…19	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ						
1								
2	BLTZAL	BGEZAL						

COPz

Bits	23…21							
25…24	0	1	2	3	4	5	6	7
0	MF		CF		MT		CT	
1	BC	†	†	†	†	†	†	†

Co-Processor Specific Operations

COP0

Bits	2…0							
4…3	0	1	2	3	4	5	6	7
0		TLBR	TLBWI				TLBWR	
1	TLBP							
2	RFE							
3								

Memory

Overview

The PSX’s memory consists of four 512k 60ns SRAM chips creating 2 megabytes of system memory. The RAM is arranged so that the addresses at 0x00xxxxxx, 0xA0xxxxxx, 0x80xxxxxx all point to the same physical memory. The PSX has a special coprocessor called cop0 that handles almost every aspect of memory management. Let us first examine how the memory looks and then how it is managed.

The PSX Memory Map

0x0000_0000- 0x0000_ffff	**Kernel (64K)**
0x0001_0000 0x001f_ffff	**User Memory (1.9 Meg)**
0x1f00_0000- 0x1f00_ffff	**Parallel Port (64K)**
0x1f80_0000- 0x1f80_03ff	**Scratch Pad (1024 bytes)**
0x1f80_1000- 0x1f80_2fff	**Hardware Registers (8K)**

0x8000_0000 **0x801f_ffff**	**Kernel and User Memory Mirror (2 Meg)** **Cached**
0xa000_0000 **0xa01f_ffff**	**Kernel and User Memory Mirror (2 Meg)** **Uncached**
0xbfc0_0000-0xbfc7_ffff	**BIOS (512K)**

All blank areas represent the absence of memory. The mirrors are used mostly for caching and exception handling purposes The Kernel is also mirrored in all three user memory spaces.

Virtual Memory

The PSX uses a memory architecture known as “Virtual Memory” to help with general system memory and cache management. In a nutshell what the PSX does is mirror the two meg of addressable space into 3 segments at three different virtual addresses. The names of these segments are Kuseg, Kseg0, and Kseg1.

Kuseg spans from 0x0000_0000 to 0x001f_ffff. This is what you might call “real” memory. This facilitates the kernel having direct access to user memory regions.

Kseg0 begins at virtual address 0x8000_0000 and goes to 0x801f_ffff. This segment is always translated to a linear 2MB region of the physical address space starting at physical address 0. All references through this segment are cacheable. When the most significant three bits of the virtual address are “100”, the virtual address resides in kseg0. The physical address is constructed by replacing these three bits of the virtual address with the value “000”.

Kseg1 is also a linear 2MB region from 0xa000_0000 to 0xa01f_ffff pointing to the same address at address 0. When the most significant three bits of the virtual address are “101”, the virtual address resides in kseg1. The physical address is constructed by replacing these three bits of the virtual address with the value “000”. Unlike kseg0, references through kseg1 are not cacheable.

Looking a little deeper into how virtual memory works, the following shows the anatomy of an R3000A virtual address. The most significant 20 bits of the 32-bit virtual address are called the virtual page number, or VPN. Only the three highest bits (segment number) are involved in the virtual to physical address translation.

31				0
			VPN	Offset
31	30	29	20	12

bits 31-29

0xx kuseg

100 kseg0

101 kseg1

The three most significant bits of the virtual address identify which virtual address segment the processor is currently referencing; these segments have associated with them the mapping algorithm to be employed, and whether virtual addresses in that segment may reside in the cache. Pages are mapped by substituting a 20-bit physical frame number (PFN) for the 20-bit virtual page number field of the virtual address. This substitution is performed through the use of the on-chip Translation Lookaside Buffer (TLB). The TLB is a fully associative memory that holds 64 entries to provide a mapping of 64 4kB pages. When a virtual reference to kuseg each TLB entry is probed to see if it maps the corresponding VPN.

Virtual to physical memory translation

The following table is a quick look at how virtual memory gets translated via the Translation Lookaside Buffer. This whole subsystem of memory management is handled by Cop0.

Cop0, The System Control Coprocessor

This Unit is actually part of the R3000A. This particular cop0 has been modified from the original R3000A cop0 architecture with the addition of a few registers and functions. Cop0 contains 16 32-bit control registers that control the various aspects of memory management, system interrupt (exception) management, and breakpoints. Much of it is compatible with the normal R3000A cop0. The following is an overview of the Cop0 registers.

Cop0 Registers

Number	Mnemonic	Name	Read/ Write	Usage
0	INDX	Index	r/w	Index to an entry in the 64-entry TLB file
1	RAND	Random	r	Provides software with a “suggested” random TLB entry to be written with the correct translation
2	TLBL	TBL low	r/w	Provides the data path for operations which read, write, or probe the TLB file (first 32 bits)
3	BPC	Breakpoint PC	r/w	Sets the breakpoint address to break on execute
4	CTXT	Context	r	Duplicates information in the BADV register, but provides this information in a form that may be more useful for a software TLB exception handler.
5	BDA	Breakpoint data	r/w	Sets the breakpoint address for load/store operations

6	**PIDMASK**	**PID Mask**	**r/w**	**Process ID mask**
7	**DCIC**	**Data/ Counter interrupt control**	**r/w**	**Breakpoint control**
8	**BADV**	**Bad Virtual Address**	**r**	**Contains the address whose reference caused an exception.**
9	**BDAM**	**Break data mask**	**r/w**	**Data fetch address is ANDed with this value and then compared to the value in BDA**
10	**TLBH**	**TBL high**	**r/w**	**Provides the data path for operations which read, write, or probe the TLB file (second 32 bits)**
11	**BPCM**	**Break point counter mask**	**r/w**	**Program counter is ANDed with this value and then compared to the value in BPC**
12	**SR**	**System status register**	**r/w**	**Contains all the major status bits**
13	**CAUSE**	**Cause**	**r**	**Describes the most recently recognized exception**
14	**EPC**	**Exception Program Counter**	**r**	**Contains the return address after an exception**
15	**PRID**	**Processor ID**	**r**	**Cop0 type and revision level**
16	**ERREG**	**???**	**?**	**????**

Note that some of these registers will be explained later in the part on exception handling. But for now we will return to how the Cop0 is used in memory management.

Returning to the TLB

As stated before the TLB is a fully associative memory that holds 64 entries to provide a mapping of 64 4kB pages. Each TLB entry is 64 bits wide. This is referenced by the Index, Random, TBL high, and TBL low. It is used to virtual to physical address mapping.

The Index Register

The Index register is a 32-bit, read-write register, which has a 6-bit field used to index to a specific entry in the 64-entry TLB file. The high-order bit of the register is a status bit which reflects the success or failure of a TLB Probe (tlbp) instruction.. The Index register also specifies the TLB entry that will be affected by the TLB Read (tlbr) and TLB Write Index (tlbwi) instructions. the following shows the format of the Index register.

31	30	14	13	8	7	0
P	0		Index		0	
1	17		6		8	

P Probe failure. Set to 1 when the last TLBProbe (tlbp) instruction was unsuccessful.

Index Index to the TLB entry that will be affected by the TLBRead and TLBWrite instructions.

0 Reserved. Must be written as zero, returns zero when read.

The Random Register

The Random register is a 32-bit read-only register. The format of the Random register is below. The six-bit Random field indexes a Random entry in the TLB. It is basically a counter which decrements on every clock cycle, but which is constrained to count in the range of 63 to 8. That is, software is guaranteed that the Random register will never index into the first 8 TLB entries. These entries can be “locked” by software into the TLB file, guaranteeing that no TLB miss exceptions will occur in operations which use those virtual address. This is useful for particularly critical areas of the operating system.

0	Random	0

18	6	8

Random A random index (with a value from 8 to 63) to a TLB entry.

0 Reserved. Returns zero when read.

The Random register is typically used in the processing of a TLB miss exception. The Random register provides software with a “suggested” TLB entry to be written with the correct translation; although slightly less efficient than a Least Recently Used (LRU) algorithm, Random replacement offers substantially similar performance while allowing dramatically simpler hardware and software management. To perform a TLB replacement, the TLB Write Random (tlbwr) instruction is used to write the TLB entry indexed by this register. At reset, this counter is preset to the value ‘63’. Thus, it is possible for two processors to operate in “lock-step”, even when using the Random TLB replacement algorithm. Also, software may directly read this register, although this feature probably has little utility outside of device testing and diagnostics.

TBL High and TBL Low Registers

These two registers provide the data path for operations which read, write, or probe the TLB file. The format of these registers is the same as the format of a TLB entry.

TBL High			TBL Low						
VPN	PID	0	FPN	N	D	V	G	0	
20	6	6	20	1	1	1	1	8	

VPN Virtual Page Number. Bits 31..12 of virtual address.

PID Process ID field. A 6-bit field which lets multiple processes share the TLB while each process has a distinct mapping of otherwise identical virtual page numbers.

PFN Page Frame Number. Bits 31..12 of the physical address.

N Non-cacheable. If this bit is set, the page is marked as non-cacheable

D Dirty. If this bit is set, the page is marked as "dirty" and therefore writable. This bit is actually a "write-protect" bit that software can use to prevent alteration of data

V Valid. If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLBL or TLBS Miss occurs.

G Global. If this bit is set, the R3000A ignores the PID match requirement for valid translation. In kseg2, the Global bit lets the kernel access all mapped data without requiring it to save or restore PID (Process ID) values.

1. **Reserved. Must be written as '0', returns '0' when read.**

Exception Handling

There are times when in is necessary to suspend a program in order to process a hardware or software function. The exception processing capability of the R3000A is provided to assure an orderly transfer of control from an executing program to the kernel. Exceptions may be broadly divided into two categories: they can be caused by an instruction or instruction sequence, including an unusual condition arising during its execution; or can be caused by external events such as interrupts. When an R3000A detects an exception, the normal sequence of instruction flow is suspended; the processor is forced to kernel mode where it can respond to the abnormal or asynchronous event. The table below lists the exceptions recognized by the R3000A.

Exception	**Mnemonic**	**Cause**
Reset	**Reset**	**Assertion of the Reset signal causes an exception that transfers control to the special vector at virtual address 0xbfc0_0000 (The start of the BIOS)**
Bus Error	**IBE** **DBE (Data)**	**Assertion of the Bus Error input during a read operation, due to such external events as bus timeout, backplane memory errors, invalid physical address, or invalid access types.**
Address Error	**AdEL (Load)**	**Attempt to load, fetch, or store an unaligned word; that is, a word or**

	AdES (Store)	**halfword at an address not evenly divisible by four or two, respectively. Also caused by reference to a virtual address with most significant bit set while in User Mode.**
Overflow	**Ovf**	**Twos complement overflow during add or subtract.**
System Call	**Sys**	**Execution of the SYSCALL Trap Instruction**
Breakpoint	**Bp**	**Execution of the break instruction**
Reserved Instruction	**RI**	**Execution of an instruction with an undefined or reserved major operation code (bits 31:26), or a special instruction whose minor opcode (bits 5:0) is undefined.**
Co-processor Unusable	**CpU**	**Execution of a co-processor instruction when the CU (Co-processor usable) bit is not set for the target co-processor.**
TLB Miss	TLBL (Load) TLBS (Store)	A referenced TLB entry's Valid bit isn't set
TLB Modified	**Mod**	**During a store instruction, the Valid bit is set but the dirty bit is not set in a matching TLB entry.**
Interrupt	**Int**	**Assertion of one of the six hardware interrupt inputs or setting of one of the two software interrupt bits in the Cause register.**

Returning to the Cop0

The Cop0 controls the exception handling with the use of the Cause register, the EPC register, the Status register, the BADV register, and the Context register. A brief description of each follows, after which the rest of the Cop0 registers for breakpoint management will be described for the sake of completeness.

The Cause Register

The contents of the Cause register describe the last exception. A 5-bit exception code indicates the cause of the current exception; the remaining fields contain detailed information specific to certain exceptions. All bits in this register, with the exception of the SW bits, are read-only.

31								0
BD	0	CE	0	IP	SW	0	EXECODE	0
1	1	2	12	6	2	1	5	2

BD Branch Delay. The Branch Delay bit is set (1) if the last exception was taken while the

processor was executing in the branch delay slot. If so, then the EPC will be rolled back to point to the branch instruction, so that it can be re-executed and the branch direction re-determined..

CE Coprocessor Error, Contains the coprocessor number if the exception occurred because of a coprocessor instruction for a coprocessor which wasn't enabled in SR.

IP Interrupts Pending. It indicates which interrupts are pending. Regardless of which interrupts are masked, the IP field can be used to determine which interrupts are pending.

SW Software Interrupts. The SW bits can be written to set or reset software interrupts. As long as any of the bits are set within the SW field they will cause an interrupt if the corresponding bit is set in SR under the interrupt mask field.

0 Reserved, Must Be Written as 0. Returns 0 when Read

EXECODE Exception Code Field. Describes the type of exception that occurred. The following table lists the type of exception that it was.

Number	Mnemonic	Description
0	INT	External Interrupt
1	MOD	TLB Modification Exception

2	**TLBL**	**TLB miss Exception (Load or instruction fetch)**
3	**TLBS**	**TLB miss exception (Store)**
4	**ADEL**	**Address Error Exception (Load or instruction fetch)**
5	**ADES**	**Address Error Exception (Store)**
6	**IBE**	**Bus Error Exception (for Instruction Fetch)**
7	**DBE**	**Bus Error Exception (for data Load or Store)**
8	**SYS**	**SYSCALL Exception**
9	**BP**	**Breakpoint Exception**
10	**RI**	**Reserved Instruction Exception**
11	**CPU**	**Co-Processor Unusable Exception**
12	**OVF**	**Arithmetic Overflow Exception**
13-31	**-**	**Reserved**

The EPC (Exception Program Counter) Register

The 32-bit EPC register contains the virtual address of the instruction which took the exception, from which point processing resumes after the exception has been serviced. When the virtual address of the instruction resides in a branch delay slot, the EPC contains the virtual address of the instruction immediately preceding the exception (that is, the EPC points to the Branch or Jump instruction).

BADV Register

The BADV register saves the entire bad virtual address for any addressing exception.

Context Register

The Context register duplicates some of the information in the BADV register, but provides this information in a form that may be more useful for a software TLB exception handler. The following illustrates the layout of the Context register. The Context register is used to allow software to quickly determine the main memory address of the page table entry corresponding to the bad virtual address, and allows the TLB to be updated by software very quickly (using a nine-instruction code sequence).

PTE Base	BADV	0
11	19	2

0 Reserved, read as 0 and must be written as 0

BADV Failing virtual page number (set by hardware read only derived from BADV register

PTE Base Base address of page table entry, set by the kernel

The Status Register

The Status register contains all the major status bits; any exception puts the system in Kernel mode. All bits in the status register, with the exception of the TS (TLB Shutdown) bit, are readable and writable; the TS bit is read-only. Figure 5.4 shows the functionality of the various bits in the status register. The status register contains a three level stack (current, previous, and old) of the kernel/user mode bit (KU) and the interrupt enable (IE) bit. The stack is pushed when each exception is taken, and popped by the Restore From Exception instruction. These bits may also be directly read or written. At reset, the SWc, KUc, and IEc bits are set to zero; BEV is set to one; and the value of the TS bit is set to 0 (TS = 0) The rest of the bit fields are undefined after reset.

31																		0
CU	0	RE	0	BEV	TS	PE	CM	PZ	SwC	IsC	IntMask	0	KUo	IEo	KUp	IEp	KUc	IEc

4	2	1	2	1	1	1	1	1	1	1	8	2	1	1	1	1	1	1

The various bits of the status register are defined as follows:

CU Co-processor Usability. These bits individually control user level access to co-processor operations, including the polling of the BrCond input port and the manipulation of the System Control Co-processor (CP0). CU2 is for the GTE, CU1 is for the FPA, which is not available in the PSX.

RE Reverse Endianness. The R3000A allows the system to determine the byte ordering convention for the Kernel mode, and the default setting for user mode, at reset time. If this bit is cleared, the endianness

defined at reset is used for the current user task. If this bit is set, then the user task will operate with the opposite byte ordering convention from that determined at reset. This bit has no effect on kernel mode.

BEV Bootstrap Exception Vector. The value of this bit determines the locations of the exception vectors of the processor. If BEV = 1, then the processor is in “Bootstrap” mode, and the exception vectors reside

in the BIOS ROM. If BEV = 0, then the processor is in normal mode, and the exception vectors reside in RAM.

TS TLB Shutdown. This bit reflects whether the TLB is functioning.

PE Parity Error. This field should be written with a "1" at boot time. Once initialized, this field will always be read as "0'.

CM Cache Miss. This bit is set if a cache miss occurred while the cache was isolated. It is useful in determining the size and operation of the internal cache subsystem.

PZ Parity Zero. This field should always be written with a "0".

SwC Swap Caches. Setting this bit causes the execution core to use the on-chip instruction cache as a data cache and vice-versa. Resetting the bit to zero unswaps the caches. This is useful for certain operations

such as instruction cache flushing. This feature is not intended for normal operation with the caches swapped.

IsC Isolate Cache. If this bit is set, the data cache is “isolated” from main memory; that is, store operations modify the data cache but do not cause a main memory write to occur, and load operations return the data value from the cache whether or not a cache hit occurred. This bit is also useful in various operations such as flushing.

IM Interrupt Mask. This 8-bit field can be used to mask the hardware and software interrupts to the execution engine (that is, not allow them to cause an exception). IM(1:0) are used to mask the software interrupts, and IM (7:2) mask the 6 external interrupts. A value of ‘0’ disables a particular interrupt, and a ‘1’ enables it. Note that the IE bit is a global interrupt enable; that is, if the IE is used to disable interrupts, the value of particular mask bits is irrelevant; if IE enables interrupts, then a particular interrupt is selectively masked by this field.

KUo Kernel/User old. This is the privilege state two exceptions previously. A ‘0’ indicates kernel mode.

IEo Interrupt Enable old. This is the global interrupt enable state two exceptions previously. A ‘1’ indicates that interrupts were enabled, subject to the IM mask.

KUp Kernel/User previous. This is the privilege state prior to the current exception A ‘0’ indicates kernel mode.

IEp Interrupt Enable previous. This is the global interrupt enable state prior to the current exception. A '1' indicates that interrupts were enabled, subject to the IM mask.

KUc Kernel/User current. This is the current privilege state. A '0' indicates kernel mode.

IEc Interrupt Enable current. This is the current global interrupt enable state. A '1' indicates that interrupts are enabled, subject to the IM mask.

0 Fields indicated as '0' are reserved; they must be written as '0', and will

return '0' when read.

PRID Register

This register is useful to software in determining which revision of the processor is executing the code. The format of this register is illustrated below.

0	Imp	Rev
16	8	8

Imp 3 CoP0 type R3000A

7 IDT unique (3041) use REV to determine correct configuration.

Rev Revision level.

EXCEPTION VECTOR LOCATIONS

The R3000A separates exceptions into three vector spaces. The value of each vector depends on the BEV (Boot Exception Vector) bit of the status register, which allows two alternate sets of vectors (and thus two different pieces of code) to be used. Typically, this is used to allow diagnostic tests to occur before the

functionality of the cache is validated; processor reset forces the value of the BEV bit to a 1.

Exception	**Virtual Address**	**Physical Address**
Reset	0xbfc0_0000	0x1fc0_0000
UTLB Miss	**0x8000_0000**	**0x0000_0000**
General	**0x8000_0080**	**0x0000_0080**

Exception Vectors When BEV = 0

Exception	**Virtual Address**	**Physical Address**
Reset	0xbfc0_0000	0x1fc0_0000
UTLB Miss	**0xbfc0_0100**	**0x1fc0_0100**
General	**0xbfc0_0180**	**0x1fc0_0180**

Exception Vectors When BEV =1

Exception Priority

The following is a priority list of exceptions:

Reset At any time (highest)

AdEL Memory (Load instruction)

AdES Memory (Store instruction)

DBE Memory (Load or store)

MOD ALU (Data TLB)

TLBL ALU (DTLB Miss)

TLBS ALU (DTLB Miss)

Ovf ALU

Int ALU

Sys RD (Instruction Decode)

Bp RD (Instruction Decode)

RI RD (Instruction Decode)

CpU RD (Instruction Decode)

TLBL I-Fetch (ITLB Miss)

AdEL IVA (Instruction Virtual Address)

IBE RD (end of I-Fetch, lowest)

Breakpoint Management

The following is a listing of the registers in Cop0 that are used for breakpoint management. These registers are very useful for low-level debugging.

BPC

Breakpoint on execute. Sets the breakpoint address to break on execute.

BDA

Breakpoint on data access. Sets the breakpoint address for load/store operations

DCIC

Breakpoint control. To use the Execution breakpoint, set PC. To use the Data access

breakpoint set DA and either R, W or both. Both breakpoints can be used simultaneously. When a breakpoint occurs the PSX jumps to 0x0000_0040.

1	1	1	0	W	R	DA	PC	1	0

1	1	1	1	1	1	1	1	1	23

W 0

1 Break on Write

R 0

1 Break on Read

DA 0 Data access breakpoint disabled

1 Data access breakpoint enabled

PC 0 Execution breakpoint disabled

1 Execution breakpoint enabled

BDAM

Data Access breakpoint mask. Data fetch address is ANDed with this value and then compared

to the value in BDA

BPCM

Execute breakpoint mask. Program counter is ANDed with this value and then compared to

the value in BPC.

DMA

From time to time the PSX will need to take the CPU off the main bus in order to give a device access directly to Memory. The devices able to take control of the bus are the CD-ROM, MDEC, GPU, SPU, and the Parallel port. There are 7 DMA channels in all (The GPU and MDEC use two) The DMA registers reside between 0x1f80_1080 and 0x1f80_10f4. The DMA channel registers are located starting at 0x1f80_1080. The base address for each channel is as follows

Base Address	**Channel Number**	**Device**
0x1f80_1080	**DMA channel 0**	**MDECin**
0x1f80_1090	**DMA channel 1**	**MDECout**
0x1f80_10a0	**DMA channel 2**	**GPU (lists + image data)**
0x1f80_10b0	**DMA channel 3**	**CD-ROM**
0x1f80_10c0	**DMA channel 4**	**SPU**
0x1f80_10d0	**DMA channel 5**	**PIO**
0x1f80_10e0	**DMA channel 6**	**GPU OTC (reverse clear the Ordering Table)**

Each channel has three 32-bit control registers at a offset of the base address for that particular channel. These registers are the DMA Memory Address Register (D_MADR) at the base address, DMA Block Control Register (D_BCR)at base+4, and the DMA Channel Control Register (D_CHCR) at base+8.

In order to use DMA the appropriate channel must be enabled. This is done using the DMA Primary Control Register (DPCR) located at 0x1f80_10f0.

DMA Primary Control Register (DPCR) 0x1f80_10f0

	DMA6	**DMA5**	**DMA4**	**DMA3**	**DMA2**	**DMA1**	**DMA0**
4	**4**	**4**	**4**	**4**	**4**	**4**	**4**

Each register has a 4 bit control block allocated in this register.

Bit 3 1= DMA Enabled

2 Unknown

1 Unknown

0 Unknown

Bit 3 must be set for a channel to operate.

As stated above, each device has three 32-bit control registers within it's own DMA address space. The following describes their functions. The n represents 8,9,a,b,c,d,e for DMA channels 0,1,2,3,4,5,6 respectively.

DMA Memory Address Register (D_MADR) 0x1f80_10n0

31	0
MADR	

MADR Pointer to the virtual address the DMA will start reading from/writing to.

DMA Block Control Register (D_BCR) 0x1f80_10n4

31	0
BA	BS

16	16

BA Amount of blocks

BS Blocksize (words)

The channel will transfer BA blocks of BS words. Take care not to set the size larger than the buffer of the corresponding unit can hold. (GPU & SPU both have a $10 word buffer). A larger blocksize, means a faster transfer.

DMA Channel Control Register (D_CHCR) 0x1f80_10n8

31							0
0	TR	0	LI	CO	0		DR
7	1	13	1	1	8		1

TR 0 No DMA transfer busy.

1 Start DMA transfer/DMA transfer busy.

LR 1 Transfer linked list. (GPU only)

CO 1 Transfer continuous stream of data.

DR 1 Direction from memory

0 Direction from memory

The last register is used to control DMA interrupts. The usage is currently unknown.

DMA Interrupt Control Register (DICR) 0x1f80_10f4

Video

Overview

The GPU is the unit responsible for the graphical output of the PSX. It handles display and drawing of all graphics. It has the control over an 1MB frame buffer, which at 16 bits per pixel gives you a maximum “surface” of 1024x512 resolution. It also contains a 2Kb texture cache for increased speed. The display can be set for 15-bit color or 24-bit color.

Because the PSX also totally lacks an FPU. A second coprocessor has been added called the Geometry Transformation Engine or GTE. The GTE is the heart of all 3d calculations on the PSX. The GTE can perform vector and matrix operations, perspective transformation, color equations and the like. It is much faster than the CPU on these operations. It is mounted as the second coprocessor (Cop2) and as such takes up no physical address space in the PSX. The GTE is covered later in the document.

The Graphics Processing Unit (GPU)

As stated before the GPU is responsible for graphical output. It has at it’s disposal a 1 MB frame buffer and registers to access it. The frame buffer it totally inaccessible to the CPU, meaning that it doesn’t reside in addressable memory. The only way to access it is through the GPU. The GPU is able to take “commands” from the CPU, or via DMA to place objects on the frame buffer to be displayed. Communication is handled through a command and data port. It has a 64 byte command FIFO buffer, which can hold up to 3 commands and is connected to a DMA channel for transfer of image data and linked command lists (channel 2) and a DMA channel for reverse clearing an Ordering Table (channel 6).

Communication and Ordering Tables (OT).

All data regarding drawing and drawing environment are sent as packets to the GPU. Each packet tells the GPU how and where to draw one primitive, or it sets one of the drawing environment parameters. The display environment is set up through single word commands using the control port of the GPU.

Packets can be forwarded word by word through the data port of the GPU, or more efficiently for large numbers of packets through DMA. A special DMA mode was created for this so large numbers of packets can be sent and managed easily. In this mode a list of packets is sent, where each entry in the list contains a header which is one word containing the address of the next entry and the size of the packet and the packet itself. A result of this is that the packets do not need to be stored sequentially. This makes it possible to easily control the order in which packets get processed. The GPU processes the packets it gets in the order they are offered. So the first entry in the list also gets drawn first. To insert a packet into the middle of the list simply find the packet after which needs it to be processed,

replace the address in that packet with the address of the new packet, and let that point to the address that was replaced.

To aid in finding a location in the list, the Ordering Table was invented. At first this is basically a linked list with entries of packet size 0, so it's a list of only list entry headers, where each entry points to to the next entry. Then as primitives are generated by your program you can then add them to the table at a certain index. Just read the address in the table entry and replace it with the address of the new packet and store the address from the table in the packet. When all packets are generated drawing will just require passing the address of the first list entry to the DMA and the packets will get drawn in the order you entered the packets to the table. Packets entered at a higher table index will get drawn after those entered at a lower table index. Packets entered at the same index will get drawn in the order they were entered, the last one first.

In 3d drawing it's most common that you want the primitives with the highest Z value to be drawn first, so it would be nice if the table would be drawn the other way around, so the Z value can be used as index. This is a simple thing, just make a table of which each entry points to the previous entry, and start the DMA with the address of the last table entry. To assist you in making such a table, a special DMA channel is available which creates it for you.

The Frame Buffer

The frame buffer is the memory which stores all graphic data which the GPU can access and manipulate, while drawing and displaying an image . The memory is under the GPU and cannot be accessed by the CPU directly. It is operated solely by the GPU. The frame buffer has a size of 1 MB and is treated as a space of 1024 pixels wide and 512 pixels high. Each "pixel" has the size of one word (16 bit). It is not treated linearly like usual memory, but is accessed through coordinates, with an upper left corner of (0,0) and a lower right corner of (1023,511).

When data is displayed from the frame buffer, a rectangular area is read from the specified coordinate within this memory. The size of this area can be chosen from several hardware defined types. Note that these hardware sizes are only valid when the X and Y stop/start registers are at their default values. This display area can be displayed in two color formats, being 15bit direct and 24bit direct. The data format of one pixel is as follows.

15-bit direct display

Pixel						
M	Blue		Green		Red	
15	14	10	9	5	4	0

This means each color has a value of 0-31. The MSB of a pixel (M) is used to mask the pixel.

24-bit direct display

The GPU can also be set to 24bit mode, in which case 3 bytes form one pixel, 1 byte for each color. Data in this mode is arranged as follows:

Pixel 0				Pixel 1				Pixel 2			
G0		R0		R1		B0		B1		G1	
15	8	7	0	15	8	7	0	15	8	7	0

Thus 2 display pixels are encoded in 3 frame buffer pixels. They are displayed as follows: [R0,G0,B0] [R1,G1,B1].

Primitives.

A basic figure which the GPU can draw is called a primitive, and it can draw the following:

- **Polygon**

The GPU can draw 3 point and 4 point polygons. Each point of the polygon specifies a point in the frame buffer. The polygon can be also be gourad shaded. The correct order of vertices for 4 point polygons is as follows

1		2

3		4

A 4 point polygon is processed internally as two 3 point polygons. also note when drawing a polygon the GPU will not draw the right most and bottom edge. So a (0,0)-(32,32) rectangle will actually be drawn as (0,0)-(31,31). Make sure adjoining polygons have the same coordinates if you want them to touch each other!.

- **Polygon with texture**

A primitive of this type is the same as above, except that a texture is applied. Each vertex of the polygon maps to a point on a texture page in the frame buffer. The polygon can be gourad shaded.

Because a 4 point polygon is processed internally as two 3 point polygons, texture mapping is also done independently for both halves. This has some annoying consequences.

- **Rectangle**

A rectangle is defined by the location of the top left corner and its width and height. Width and height can be either free, 8*8 or 16*16. It's drawn much faster than a polygon, but gourad shading is not possible.

- **Sprite**

A sprite is a textured rectangle, defined as a rectangle with coordinates on a texture page. Like the rectangle is drawn much faster than the polygon equivalent. No gourad shading possible. Even though the primitive is called a sprite, it has nothing in common with the traditional sprite, other than that it's a rectangular piece of graphics. Unlike the PSX sprite, the traditional sprite is NOT drawn to the

bitmap, but gets sent to the screen instead of the actual graphics data at that location at display time.

- **Line**

A line is a straight line between 2 specified points. The line can be gourad shaded. A special form is the polyline, for which an arbitrary number of points can be specified.

- **Dot**

The dot primitive draws one pixel at the specified coordinate and in the specified color. It is actually a special form of rectangle, with a size of 1x1.

Textures

A texture is an image put on a polygon or sprite. It is necessary to prepare the data beforehand in the frame buffer. This image is called a texture pattern. The texture pattern is located on a texture page which has a standard size and is located somewhere in the frame buffer, see below. The data of a texture can be stored in 3 different modes

- **15-bit direct mode**

I0						
S	Blue		Green		Red	
15	14	10	9	5	4	0

This means each color has a value of 0-31. The MSB of a pixel (S) is used to specify it the pixel is semi transparent or not. More on that later.

- **8bit CLUT mode,**

Each pixel is defined by 8bits and the value of the pixel is converted to a 15-bit color using the CLUT(color lookup table) much like standard VGA pictures. So in effect you have 256 colors which are in 15bit precision.

I1		I0	
15	8	7	0

I0 is the index to the CLUT for the left pixel, I1 for the right.

- **4-bit CLUT mode,**

Same as above except that only 16 colors can be used. Data is arranged as follows:

I3		I2		I1		I0	
15	12	11	8	7	4	3	0

I0 is first drawn to the left to I3 to the right.

- **Texture Pages**

Texture pages have a unit size of 256*256 pixels, regardless of color mode. This means that in the frame buffer they will be 64 pixels wide for 4bit CLUT, 128 pixels wide for 8bit CLUT and 256 pixels wide for 15-bit direct. The pixels are addressed with coordinates relative to the location of the texture page, not the frame buffer. So the top left texture coordinate on a texture page is (0,0) and the bottom right one is (255,255). The pages can be located in the frame buffer on X multiples of 64 and Y multiples of 256. More than one texture page can be set up, but each primitive can only contain texture from one page.

- **Texture Windows**

The area within a texture window is repeated throughout the texture page. The data is not actually stored all over the texture page but the GPU reads the repeated patterns as if they were there. The X and Y

and H and W must be multiples of 8.

- **CLUT (Color Lookup Table)**

The CLUT is a the table where the colors are stored for the image data in the CLUT modes. The pixels of those images are used as indexes to this table. The CLUT is arranged in the frame buffer as a 256x1 image for the 8bit CLUT mode, and a 16x1 image for the 4bit CLUT mode. Each pixel as a 16 bit value, the first 15 used of a 15 bit color, and the 16th used for semi-transparency. The CLUT data can be arranged in the frame buffer at X multiples of 16 (X=0,16,32,48,etc) and anywhere in the Y range of 0-511. More than one CLUT can be prepared but only one can be used for each

primitive.

- **Texture Caching**

If polygons with texture are displayed, the GPU needs to read these from the frame buffer. This slows down the drawing process, and as a result the number of polygons that can be drawn in a given time span. To speed up this process the GPU is equipped with a texture cache, so a given piece of texture needs not to be read multiple times in succession. The texture cache size depends on the color mode used for the textures. In 4-bit CLUT mode it has a size of 64x64, in 8-bit CLUT it's 32x64 and in 15-bit direct is 32x32. A general speed up can be achieved by setting up textures according to these sizes. For further speed gain a more precise knowledge of how the cache works is necessary.

Cache blocks

The texture page is divided into non-overlapping cache blocks, each of a unit size according to color mode. These cache blocks are tiled within the texture page.

Cache Block 0	1	2...

- **Cache entries**

Each cache block is divided into 256 cache entries, which are numbered sequentially, and are 8 bytes wide. So a cache entry holds 16 4-bit CLUT pixels 8 8-bit CULT pixels, or 4 15bitdirect pixels.

4-bit and 8-bit CLUT

0	1	2	3
4	5	6	7
8	9	...	
c			

15-bit direct

0	1	2	3	4	5	6	7
8	9	a	b	c	d	e	f
10	11	...	...	...			
18	...						

The cache can hold only one cache entry by the same number, so if for example, a

piece of texture spans multiple cache blocks and it has data on entry 9 of block 1, but also on entry 9 of block 2, these cannot be in the cache at once.

Rendering options

There are 3 modes which affect the way the GPU renders the primitives to the frame buffer.

- **Semi Transparency**

When semi transparency is set for a pixel, the GPU first reads the pixel it wants to write to, and then calculates the color it will write from the 2 pixels according to the semi-transparency mode selected. Processing speed is lower in this mode because additional reading and calculating are necessary. There are 4 semi-transparency modes in the GPU.

B= the pixel read from the image in the frame buffer, F = the half transparent pixel

- **1.0 x B + 0.5 x F**
- **1.0 x B + 1.0 x F**
- **1.0 x B - 1.0 x F**
- **1.0 x B + 0.25 x F**

A new semi transparency mode can be set for each primitive. For primitives without texture semi- transparency can be selected. For primitives with texture semi transparency is stored in the MSB of each pixel, so some pixels can be set to STP others can be drawn opaque. For the CLUT modes the STP bit is obtained from the CLUT. So if a color index points to a color in the CLUT with the MSB set, it will be drawn semi transparent.

When the color is black(BGR=0), STP is processed different from when it's not black (BGR<>0). The table below shows the differences:

Transparency Processing (bit 1 of command packet)			
BGR	**STP**	**off**	**on**
0,0,0	**0**	**Transparent**	**Transparent**
0,0,0	**1**	**Non-transparent**	**Non-transparent**
x,x,x	**0**	**Non-transparent**	**Non-transparent**
x,x,x	**1**	**Non-transparent**	**Transparent**

- **Shading**

The GPU has a shading function, which will scale the color of a primitive to a specified brightness. There are 2 shading modes: Flat shading, and gourad shading. Flat shading is the mode in which one brightness value is specified for the entire primitive. In gourad shading mode, a different brightness value can be given for each vertex of a primitive, and the brightness between these points is automatically interpolated.

- **Mask**

The mask function will prevent to GPU to write to specific pixels when drawing in the frame buffer. This means that when the GPU is drawing a primitive to a masked area, it will first read the pixel at the coordinate it wants to write to, check if it's masking bit is set, and if so refrain from writing to that particular pixel. The masking bit is the MSB of the pixel, just like the STP bit. To set this masking bit, the GPU provides a mask out mode, which will set the MSB of any pixel it writes. If both mask out and mask evaluation are on, the GPU will not draw to pixels with set MSB's, and will draw pixels with set MSB's to the others, these in turn becoming masked pixels.

Drawing Environment

The drawing environment specifies all global parameters the GPU needs for drawing primitives.

- Drawing offset.

This locates the top left corner of the drawing area. Coordinates of primitives originate to this point. So if the drawing offset is (0,240) and a vertex of a polygon is located at (16,20) it will be drawn to the frame buffer at (0+16,240+20).

- Drawing clip area

This specifies the maximum range the GPU draws primitives to. So in effect it specifies the top left and bottom right corner of the drawing area.

- Dither enable

When dither is enabled the GPU will dither areas during shading. It will process internally in 24 bit and dither the colors when converting back to 15-bit. When it is off, the lower 3 bits of each color simply get discarded.

- Draw to display enable.

This will enable/disable any drawing to the area that is currently displayed.

- Mask enable

When turned on any pixel drawn to the frame buffer by the GPU will have a set masking bit. (= set MSB)

- Mask judgement enable

Specifies if the mask data from the frame buffer is evaluated at the time of drawing.

Display Environment.

This contains all information about the display, and the area displayed.

- **Display area in frame buffer**

This specifies the resolution of the display. The size can be set as follows:

Width: 256,320,384,512 or 640 pixels

Height: 240 or 480 pixels

These sizes are only an indication on how many pixels will be displayed using a default start end. These settings only specify the resolution of the display.

- **Display start/end.**

Specifies where the display area is positioned on the screen, and how much data gets sent to the screen. The screen sizes of the display area are valid only if the horizontal/vertical start/end values are default. By changing these you can get bigger/smaller display screens. On most TV's there is some black around the edge, which can be utilized by setting the start of the screen earlier and the end later. The size of the pixels is NOT changed with these settings, the GPU simply sends more data to the screen. Some monitors/TVs have a smaller display area and the extended size might not be visible on those sets.(Mine is capable of about 330 pixels horizontal, and 272 vertical in 320*240 mode)

- **Interlace enable**

When enabled the GPU will display the even and odd lines of the display area alternately. It is necessary to set this when using 480 lines as the number of scan lines on a TV screen are not sufficient to display 480 lines.

- **15bit/24bit direct display**

Switches between 15bit/24bit display mode.

- **Video mode**

Selects which video mode to use, which are either PAL or NTSC.

GPU operation

- **GPU control registers.**

There are 2 32 bit IO ports for the GPU, which are at 0x1f80_1810 for GPU Data and 0x1f80_1814 for GPU control/Status. The data register is used to exchange data with the GPU and the control/status register gives the status of the GPU when read, and sets the control bits when written to.

Control/Status Register 0x1f80_1814

Status (Read) High

31									16				
lcf	dma	com	img	busy	?	?	den	isinter	isrgb24	Video	Height	Width0	Width1
1	2	1	1	1	1	1	1	1	1	1	1	2	1

W0 W1

Width 00 0 256 pixels

01 0 320

10 0 512

11 0 640

00 1 384

Height 0 240 pixels

1 480

Video 0 NTSC

1 PAL

isrgb24 0 15-bit direct mode

1 24-bit direct mode

isinter 0 Interlace off

1 Interlace on

den 0 Display enabled

1 Display disabled

busy 0 GPU is Busy (i.e. drawing primitives)

1 GPU is Idle

img 0 Not Ready to send image (packet $c0)

1 Ready

com 0 Not Ready to receive commands

1 Ready

dma 00 DMA off, communication through GP0

01 Unknown

10 DMA CPU -> GPU

11. DMA GPU -> CPU

lcf 0 Drawing even lines in interlace mode

1 Drawing uneven lines in interlace mode

Status (Read) Low

15						0				
?	?	?	me	md	dfe	dtd	tp	abr	ty	tx

1	1	1	1	1	1	1	2	2	1	4

tx 0 0 Texture page X = tx*64

1 64

2 128

3 196

4 ...

ty 0 0 Texture page Y

1 256

abr 00 0.5xB+0.5 x F Semi transparent state

01 1.0xB+1.0 x F

10 1.0xB-1.0 x F

11 1.0xB+0.25 x F

tp 00 4-bit CLUT Texture page color mode

01 8-bit CLUT

10 15-bit

dtd 0 Dither off

1 Dither on

dfe 0 off Draw to display area prohibited

1 on Draw to display area allowed

md 0 off Do not apply mask bit to drawn pixels

1 on Apply mask bit to drawn pixels

me 0 off Draw over pixel with mask set

1 on No drawing to pixels with set mask bit.

Control (Write)

A control command is composed of one word as follows:

command		parameter	
31	16	15	0

The composition of the parameter is different for each command.

- **Reset GPU**

command 0x00

parameter 0x000000

Description Resets the GPU. Also turns off the screen. (sets status to $14802000)

- **Reset Command Buffer**

command 0x01

parameter 0x000000

Description Resets the command buffer.

- **Reset IRQ**

command 0x02

parameter 0x000000

Description Resets the IRQ.

- **Display Enable**

command 0x03

parameter 0x000000 Display disable

0x000001 Display enable

description Turns on/off display. Note that a turned off screen still gives the flicker of NTSC on a pal screen if NTSC mode is selected..

- **DMA setup.**

command 0x04

parameter 0x000000 DMA disabled

0x000001 Unknown DMA function

0x000002 DMA CPU to GPU

0x000003 DMA GPU to CPU

description Sets DMA direction.

- **Start of display area**

command 0x05

parameter bit 0x00-0x09 X (0-1023)

bit 0x0a-0x12 Y (0-512) = Y<<10 + X

description Locates the top left corner of the display area.

- **Horizontal Display range**

command 0x06

parameter bit 0x00-0x0b X1 (0x1f4-0xCDA)

bit 0x0c-0x17 X2 = X1+X2<<12

description Specifies the horizontal range within which the display area is displayed. The display is relative to the display start, so X coordinate 0 will be at the value in X1. The display end is not relative to the display start. The number of pixels that get sent to the screen in 320 mode are (X2-X1)/8. How many actually are visible depends on your TV/monitor. (normally $260-$c56)

- **Vertical Display range**

command 0x07

parameter bit 0x00-0x09 Y1

bit 0x0a-0x14 Y2 = Y1+Y2<<10

description Specifies the vertical range within which the display area is displayed. The display is relative to the display start, so Y coordinate 0 will be at the value in Y1. The display end is not relative to the display start. The number of pixels that get sent to the display are Y2-Y1, in 240 mode. (Not sure about the default values, should be something like NTSC $010-$100, PAL $023-$123)

- **Display mode**

command 0x08

parameter bit 0x00-0x01 Width 0

bit 0x02 Height

bit 0x03 Video mode: See above

bit 0x04 Isrgb24

bit 0x05 Isinter

bit 0x06 Width1

bit 0x07 Reverse flag

description Sets the display mode.

- **Unknown**

command 0x09

parameter 0x000001 ??

description Used with value $000001

- **GPU Info**

command 0x10

parameter 0x000000

0x000001

0x000002

0x000003 Draw area top left

0x000004 Draw area bottom right

0x000005 Draw offset

0x000006

0x000007 GPU Type, should return 2 for a standard GPU description. Returns requested info. Read result from GP0. 0,1 seem to return draw area top left also 6 seems to return draw offset too.

- **?????**

command 0x20

parameter ???????

description Used with value $000504

Command Packets, Data Register

Primitive command packets use an 8 bit command value which is present in all packets. They contain a 3 bit type block and a 5 bit option block of which the meaning of the bits depend on the type. layout is as follows:

Type

000 GPU command

001 Polygon primitive

010 Line primitive

011 Sprite primitive

100 Transfer command

111 Environment command

Configuration of the option blocks for the primitives is as follows:

Polygon							
Type			Option				
0	0	1	IIP	VTX	TME	ABE	TGE
7	6	5	4	3	2	1	0

Line							
Type			Option				
0	1	0	IIP	PLL	0	ABE	0
7	6	5	4	3	2	1	0

Sprite						
Type			Option			
1	0	0	Size	TME	ABE	0

7	6	5	4	3	2	1	0

IIP 0 Flat Shading

1 Gourad Shading

VTX 0 3 vertex polygon

1 4 vertex polygon

TME 0 Texture mapping off

1 Texture mapping on

ABE 0 Semi transparency off

1 Semi transparency on

TGE 0 Brightness calculation at time of texture mapping on

1 off. (draw texture as is)

Size 00 Free size (Specified by W/H)

01 1 x 1

10 8 x 8

11 16 x 16

PLL 0 Single line (2 vertices)

1 Polyline (n vertices)

- **Color information**

Color information is forwarded as 24-bit data. It is parsed to 15-bit by the GPU.

Layout as follows:

Blue		Green		Red	
23	16	15	8	7	0

- **Shading information.**

For textured primitive shading data is forwarded by this packet. Layout is the same as for color data, the RGB values controlling the brightness of the individual colors ($00-$7f). A value of $80 in a color will take the former value as data.

Blue		Green		Red	
23	16	15	8	7	0

*Texture Page information

The Data is 16 bit wide, layout is as follows:

0							TP		ABR		TY	TX			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

TX 0-0xf X*64 t texture page x coordinate

TY 0 0 texture page y coordinate

1 256

ABR 0 0.5xB+0.5 x F Semi transparency mode

1 1.0xB+1.0 x F

2 1.0xB-1.0 x F

3 1.0xB+0.25 x F

TP 0 4-bit CLUT

1 8-bit CLUT

2 15-bit direct

- **CLUT-ID**

Specifies the location of the CLUT data. Data is 16-bits.

Y coordinate 0-511								X coordinate X/16							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Abbreviations in packet list

BGR Color/Shading info see above.

xn,yn 16 bit values of X and Y in frame buffer.

un,vn 8 bit values of X and Y in texture page

tpage texture page information packet, see above

clut CULT ID, see above.

Packet list

The packets sent to the GPU are processed as a group of data, each one word wide. The data must be written to the GPU data register ($1f801810) sequentially. Once all data has been received, the GPU

starts operation.

Overview of packet commands:

- **Primitive drawing packets**

0x20 monochrome 3 point polygon

0x24 textured 3 point polygon

0x28 monochrome 4 point polygon

0x2c textured 4 point polygon

0x30 gradated 3 point polygon

0x34 gradated textured 3 point polygon

0x38 gradated 4 point polygon

0x3c gradated textured 4 point polygon

0x40 monochrome line

0x48 monochrome polyline

0x50 gradated line

0x58 gradated line polyline

0x60 rectangle

0x64 sprite

0x68 dot

0x70 8*8 rectangle

0x74 8*8 sprite

0x78 16*16 rectangle

0x7c 16*16 sprite

- **GPU command & Transfer packets**

0x01 clear cache

0x02 frame buffer rectangle draw

0x80 move image in frame buffer

0xa0 send image to frame buffer

0xc0 copy image from frame buffer

- **Draw mode/environment setting packets**

0xe1 draw mode setting

0xe2 texture window setting

0xe3 set drawing area top left

0xe4 set drawing area bottom right

0xe5 drawing offset

0xe6 mask setting

Packet Descriptions

- **Primitive Packets**

0x20 monochrome 3 point polygon									
Order	**31**	**24**	**23**	**16**	**15**	**8**	**7**	**0**	
1	**0x20**		**BGR**						**Command + Color**
2	**y0**				**x0**				**Vertex 0**
3	**y1**				**x1**				**Vertex 1**

4	y2	x2	Vertex 2

0x24 textured 3 point polygon									
Order	31	24	23	16	15	8	7	0	
1	0x24		BGR						Command + Color
2	y0				x0				Vertex 0
3	CLUT				v0		u0		CULT ID + texture coordinates vertex 0
4	y1				x1				Vertex 1
5	tpage				v1		u1		Texture page + texture coordinates vertex 1
6	y2				x2				Vertex 1
7					v2		u2		Texture coordinates vertex 2

0x28 monochrome 4 point polygon									
Order	31	24	23	16	15	8	7	0	
1	0x28		BGR						Command + Color
2	y0				x0				Vertex 0
3	y1				x1				Vertex 1

4	y2	x2	Vertex 2
5	y3	y3	Vertex 3

0x2c textured 3 point polygon									
Order	31	24	23	16	15	8	7	0	
1	0x2c		BGR						Command + Color Vertex 0
2	y0				x0				Vertex 0
3	CLUT				v0		u0		CULT ID + texture coordinates vertex 0
4	y1				x1				Vertex 1
5	tpage				v1		u1		Texture page + texture coordinates vertex 1
6	y2				x2				Vertex 2
7					v2		u2		Texture coordinates vertex 2
8	y3				x3				Vertex 3
9					v3		v3		Texture coordinates vertex 3

0x30 gradated 3 point polygon									
Order	31	24	23	16	15	8	7	0	
1	0x30		BGR0						Command + Color Vertex 0

2	**y0**	**x0**	**Vertex 0**
3		**BGR1**	**Color Vertex 1**
4	**y1**	**x1**	**Vertex 1**
5		**BGR2**	**Color Vertex 2**
6	**y2**	**x2**	**Vertex 2**

0x34 shaded textured 3 point polygon									
Order	**31**	**24**	**23**	**16**	**15**	**8**	**7**	**0**	
1	**0x34**		**BGR0**						**Command + Color Vertex 0**
2	**y0**				**x0**				**Vertex 0**
3	**CLUT**				**v0**		**u0**		**CULT ID + texture coordinates vertex 0**
4			**BGR1**						**Color Vertex 1**
5	**y1**				**x1**				**Vertex 1**
6	**tpage**				**v1**		**u1**		**Texture page + texture coordinates vertex 1**
7			**BGR2**						**Color vertex 2**
8	**y2**				**x2**				**Vertex 2**
9					**v2**		**u2**		**CULT ID + texture coordinates vertex 2**

0x38 gradated 4 point polygon									
Order	31	24	23	16	15	8	7	0	
1	0x38		BGR0						Command + Color Vertex 0
2	y0				x0				Vertex 0
3			BGR1						Color Vertex 1
4	y1				x1				Vertex 1
5			BGR2						Color Vertex 2
6	y2				x2				Vertex 2
7			BGR3						Color Vertex 3
8	y3				x3				Vertex 3

0x3c shaded textured 4 point polygon									
Order	31	24	23	16	15	8	7	0	
1	0x3c		BGR0						Command + Color Vertex 0
2	y0				x0				Vertex 0
3	CLUT				v0		u0		CULT ID + texture coordinates vertex 0

4		BGR1			Color Vertex 1
5	y1		x1		Vertex 1
6	tpage		v1	u1	Texture page + texture coordinates vertex 1
7		BGR2			Color vertex 2
8	y2		x2		Vertex 2
9			v2	u2	CULT ID + texture coordinates vertex 2
10		BGR3			Color vertex 3
11	y3		x3		Vertex 3
12			v3	32	CULT ID + texture coordinates vertex 3

0x40 monochrome line									
Order	31	24	23	16	15	8	7	0	
1	0x40		BGR						Command + Color
2	y0				x0				Vertex 0
3	y1				x1				Vertex 1

0x48 single color polyline									
Order	31	24	23	16	15	8	7	0	
1	0x48		BGR						Command + Color
2	y0				x0				Vertex 0
3	y1				x1				Vertex 1
4	y2				x2				Vertex 2
...	yn				xn				Vertex n
...	0x55555555								Termination code

Any number of points can be entered, end with termination code.

0x50 gradated line									
Order	31	24	23	16	15	8	7	0	
1	0x50		BGR0						Command + Color Vertex 0
2	y0				x0				Vertex 0
3			BGR1						Color Vertex 1
4	y1				x1				Vertex 1

0x58 gradated polyline									
Order	31	24	23	16	15	8	7	0	
1	0x58		BGR0						Command + Color Vertex 0
2	y0				x0				Vertex 0
3			BGR1						Color Vertex 1
4	y1				x1				Vertex 1
5			BGR2						Color Vertex 2
6	y2				x2				Vertex 2
...			BGRn						Color Vertex n
...	yn				xn				Vertex n
...	0x55555555								Termination code

Any number of points can be entered, end with termination code.

0x60 Rectangle									
Order	31	24	23	16	15	8	7	0	
1	0x60		BGR						Command + Color
2	y				x				upper left corner location
3	h				w				height and width

0x64 Sprite									
Order	31	24	23	16	15	8	7	0	
1	0x64		BGR						Command + Color
2	y				x				upper left corner location
3	CLUT				v		u		CULT ID + texture coordinates page y,x
4	h				w				height and width

0x68 Dot									
Order	31	24	23	16	15	8	7	0	
1	0x68		BGR						Command + Color
2	y				x				location

0x70 8x8 Rectangle									
Order	31	24	23	16	15	8	7	0	

1	0x70		BGR						Command + Color
2	y				x				location

0x74 8x8 Sprite									
Order	31	24	23	16	15	8	7	0	
1	0x74		BGR						Command + Color
2	y				x				location
3	CLUT				v		u		CULT ID + texture coordinates page y,x

0x78 16x16 Rectangle									
Order	31	24	23	16	15	8	7	0	
1	0x78		BGR						Command + Color
2	y				x				location

0x7c 16x16 Sprite	

Order	31	24	23	16	15	8	7	0	
1	0x74		BGR						Command + Color
2	y				x				location
3	CLUT				v		u		CULT ID + texture coordinates page y,x

GPU command & Transfer packets

0x01Clear cache									
Order	31	24	23	16	15	8	7	0	
1	0x01		0						clear cache

0x02 frame buffer rectangle draw									
Order	31	24	23	16	15	8	7	0	
1	0x02		BGR						Command + Color
2	y				x				upper left corner location
3	h				w				height and width

Fills the area in the frame buffer with the value in RGB. This command will draw without regard to drawing environment settings. Coordinates are absolute frame buffer coordinates. Max width is 0x3ff, max height is 0x1ff.

0x80 Rectangle									
Order	**31**	**24**	**23**	**16**	**15**	**8**	**7**	**0**	
1	**0x80**		**BGR**						**Command + Color**
2	**sy**				**sx**				**Source coordinate.**
4	**dy**				**dx**				**Destination coordinate**
5	**h**				**w**				**height and width of transfer**

Copies data within frame buffer

0x01 0xa0 send image to frame buffer									
Order	**31**	**24**	**23**	**16**	**15**	**8**	**7**	**0**	
1	**0x01**								**Reset command buffer (write to GP1 or GP0)**
2	**0xa0**		**BGR**						**Command + Color**
3	**y**				**x**				**Destination coordinate**
4	**h**				**w**				**height and width of transfer**
5	**pix1**				**pix0**				**image data**
6..									

...	**pixn**	**pixn-1**	

Transfers data from main memory to frame buffer If the number of pixels to be sent is odd, an extra should be sent. (32 bits per packet)

0x01 0xc0 send image to frame buffer									
Order	**31**	**24**	**23**	**16**	**15**	**8**	**7**	**0**	
1	**0x01**								**Reset command buffer (write to GP1 or GP0)**
2	**0xc0**		**BGR**						**Command + Color**
3	**y**				**x**				**Destination coordinate**
4	**h**				**w**				**height and width of transfer**
5	**pix1**				**pix0**				**image data**
6..									
...	**pixn**				**pixn-1**				

Transfers data from frame buffer to main memory. Wait for bit 27 of the status register to be set before reading the image data. When the number of pixels is odd, an extra pixel is read at the end.(because on packet is 32 bits)

Draw mode/environment setting packets

Some of these packets can also be by primitive packets, in any case it is the last packet of either that the GPU received that is active. so if a primitive sets tpage info, it will over write the existing data, even if it was sent by an 0xe? packet.

0xe1 draw mode setting												
31	24	23	11	10	9	8	7	6	5	4	3	0
0xe1				dfe	dtd	tp		abr		ty	tx	

See above for explanations

It seems that bits 11-13 of the status register can also be passed with this command on some GPU's other than type 2. (i.e. Command 0x10000007 doesn't return 2)

0xe2 texture window setting											
31	24	23	20	19	15	14	10	9	5	4	0
0xe2				twx		twy		tww		twh	

twx Texture window X, (twx*8)

twy Texture window Y, (twy*8)

tww Texture window width, 256-(tww*8)

twh Texture window height, 256-(twh*8)

0xe3 set drawing area top left							
31	24	23	16	15	8	7	0
0xe3				Y		X	

Sets the drawing area top left corner. X &Y are absolute frame buffer coordinates.

0xe4 set drawing area bottom right							
31	24	23	16	15	8	7	0
0xe4				Y		X	

Sets the drawing area bottom right corner. X &Y are absolute frame buffer coordinates.

0xe5 drawing offset							
31	24	23	20	13	11	10	0
0xe5				OffsY		OffsX	

Offset Y = y << 11

Sets the drawing area offset within the drawing area. X&Y are offsets in the frame buffer.

0xe6 drawing offset					
31	24	23	2	1	0

0xe6		Mask2	Mask1

Mask1 Set mask bit while drawing. 1 = on

Mask2 Do not draw to mask areas. 1= on

While mask1 is on, the GPU will set the MSB of all pixels it draws. While mask2 is on, the GPU will not write to pixels with set MSB's

DMA and the GPU

The GPU has two DMA channels allocated to it. DMA channel 2 is used to send linked packet lists to the GPU and to transfer image data to and from the frame buffer. DMA channel 6 is sets up an empty linked list, of which each entry points to the previous (i.e. reverse clear an OT.)

DMA Second Memory Address Register (D2_MADR) 0x1f80_10a0

31	0
MADR	

MADR Pointer to the virtual address the DMA will start reading from/writing to.

DMA Second Block Control Register (D2_BCR) 0x1f80_10a4

31	0
BA	BS
16	16

BA Amount of blocks

BS Block size (words)

Sets up the DMA blocks. Once started the DMA will send BA blocks of BS words. Don't set a block size larger then $10 words, as the command buffer of the GPU is 64 bytes.

DMA Second Channel Control Register (D2_CHCR) 0x1f80_10a8

31						0
0	TR	0	LI	CO	0	DR
7	1	13	1	1	8	1

TR 0 No DMA transfer busy.

1 Start DMA transfer/DMA transfer busy.

LR 1 Transfer linked list. (GPU only)

CO 1 Transfer continuous stream of data.

DR 1 Direction from memory

0 Direction from memory

This configures the DMA channel. The DMA starts when bit 18 is set. DMA is finished as soon as bit 18 is cleared again. To send or receive data to/from VRAM send the appropriate GPU packets first (0xa0/0xc0)

DMA Sixth Memory Address Register (D6_MADR) 0x1f80_10e0

31	0
MADR	

MADR Pointer to the virtual address if the last entry.

DMA Sixth Block Control Register (D6_BCR) 0x1f80_10e4

31	0
BC	

BC Number of list entries.

DMA Sixth Channel Control Register (D6_CHCR) 0x1f80_10e8

31						**0**
0	**TR**	**0**	**LI**	**CO**	**0**	**DR**
7	**1**	**13**	**1**	**1**	**8**	**1**

TR 0 No DMA transfer busy.

1 Start DMA transfer/DMA transfer busy.

LR 1 Transfer linked list. (GPU only)

CO 1 Transfer continuous stream of data.

DR 1 Direction from memory

0 Direction from memory

This configures the DMA channel. The DMA starts when bit 18 is set. DMA is finished as soon as bit 18 is cleared again. To send or receive data to/from VRAM send the appropriate GPU packets first (0xa0/0xc0) When this register is set to $11000002, the DMA channel will create an empty linked list of D6_BCR entries ending at the address in D6_MADR. Each entry has a size of 0, and points to the previous. The first entry is So if D6_MADR = $80100010, D6_BCR=$00000004, and the DMA is kicked this mwill result in a list looking like this:

0x8010_0000 0x00ff_ffff

0x8010_0004 0x0010_0000

0x8010_0008 0x0010_0004

0x8010_000c 0x0010_0008

0x8010_0010 0x0010_000c

DMA Primary Control Register (DPCR) 0x1f80_10f0

	DMA6	DMA5	DMA4	DMA3	DMA2	DMA1	DMA0
4	4	4	4	4	4	4	4

Each register has a 4 bit control block allocated in this register.

Bit 3 1= DMA Enabled

2 Unknown

1 Unknown

0 Unknown

Bit 3 must be set for a channel to operate.

Common GPU functions, step by step.

- **Initializing the GPU.**

First thing to do when using the GPU is to initialize it. To do that take the following steps:

1 - Reset the GPU (GP1 command $00). This turns off the display as well.

2 - Set horizontal and vertical start/end. (GP1 command $06, $07)

3 - Set display mode. (GP1 command $08)

4 - Set display offset. (GP1 command $05)

5 - Set draw mode. (GP0 command $e1)

6 - Set draw area. (GP0 command $e3, $e4)

7 - Set draw offset. (GP0 command $e5)

8 - Enable display.

- **Sending a linked list.**

The normal way to send large numbers of primitives is by using a linked list DMA transfer. This list is built up of entries of which each points to the next. One entry looks like this:

dw $nnYYYYYY ; nn = the number of words in the list entry

; YYYYYY = address of next list entry & 0x00ff_ffff

1 dw .. ; here goes the primitive.

2 dw ;

... dw .. ;

nn-1 dw .. ;

nn dw .. ;

The last entry in the list should have 0xffffff as pointer, which is the terminator. As soon as this value is found DMA is ended. If the entry size is set to 0, no data will be transferred to the GPU and the next entry is processed.

To send the list do this:

1 - Wait for the GPU to be ready to receive commands. (bit $1c == 1)

2 - Enable DMA channel 2

3 - Set GPU to DMA CPU->GPU mode. ($04000002)

3 - Set D2_MADR to the start of the list

4 - Set D2_BCR to zero.

5 - Set D2_CHCR to link mode, memory->GPU and DMA enable. ($01000401)

- **Uploading Image data through DMA.**

To upload an image to VRAM take the following steps:

1 - Wait for the GPU to be idle and DMA to finish. Enable DMA channel 2 if necessary.

2 - Send the 'Send image to VRAM' primitive. (You can send this through DMA if you want. Use the linked list method described above)

3 - Set DMA to CPU->GPU ($04000002) (if you didn't do so already in the previous step)

4 - Set D2_MADR to the start of the list

5 - Set D2_BCR with: bits 31-16 = Number of words to send (H*W /2)

bits 15- 0 = Block size of 1 word. ($01)

if H*W is odd, add 1. (Pixels are 2 bytes, send an extra blank pixel in case of an odd amount)

6 - Set D2_CHCR to continuous mode, memory -> GPU and DMA enable. ($01000201)

Note that H, W, X and Y are always in frame buffer pixels, even if you send image data in other formats.

You can use bigger block sizes if you need more speed. If the number of words to be sent is not a multiple of the block size, you'll have to send the remainder separately, because the GPU only accepts an extra halfword if the number of pixels is odd. (i.e. of the last word sent, only the low half word is used.) Also take care not to use block sizes bigger than 0x10, as the buffer of the GPU is only 64 bytes (=0x10 words).

- **Waiting to send commands**

You can send new commands as soon as DMA has ceased and the GPU is ready.

1 - Wait for bit $18 to become 0 in D2_CHCR

2 - Wait for bit $1c to become 1 in GP1.

The Geometry Transformation Engine (GTE)

The Geometry Transformation Engine (GTE) is the heart of all 3D calculations on the PSX. The GTE can perform vector and matrix operations, perspective transformation, color equations and the like. It is much faster than the CPU on these operations. It is mounted as the second coprocessor and as such is no physical address in the memory of the PSX. All control is done through special instructions.

Basic mathematics

The GTE is basicly an engine for vector mathematics. The basic representation of a point(vertex) in 3d space is through a vector of the sort [X,Y,Z]. In GTE operation there's basicly two kinds of these, vectors of variable length and vectors of a unit length of 1.0, called normal vectors. The first is used to decribe a locations and translations in 3d space, the second to describe a direction.

Rotation of vertices is performed by multiplying the vector of the vertex with a rotation matrix. The rotation matrix is a 3x3 matrix consisting of 3 normal vectors which are orthogonal to each other. (It's actually the matrix which describes the coordinate system in which the vertex is located in relation to the unit coordinate system. See a math book for more details.) This matrix is derived from rotation angles as follows:

s*n* = sin(*n*), c*n* = cos(*n*)

Rotation angle A about X axis:

```
| 1 0 0|
| 0 cA -sA|
| 0 sA cA|
```

Rotation angle B about Y axis:

```
| cB 0 sB|
| 0 1 0|
|-sB 0 cB|
```

Rotation angle C about Z axis:

```
| cC -sC 0|
| sC cC 0|
| 0 0 1|
```

Rotation about multiple axis can be done by multiplying these matrices with eachother. Note that the order in which this multiplication is done *IS* important. The GTE has no sine or cosine functions, so the calculation of these must be done by the CPU.

Translation is the simple addition of two vectors, relocating the vertex within its current coordinate system. Needless to say the order in which translation and rotation occur for a vector is important.

Brief Function descriptions

RTPS/RTPT

Rotate, translate and perpective transformation.

These two functions perform the final 3d calculations on one or three vertices at once. The points are first multiplied with a rotation matrix(R), and after that translated(TR). Finally a perspective transformation is applied, which results in 2d screen coordinates. It also returns an interpolation value to be used with the various depth cueing instructions.

MVMVA

Matrix & Vector multiplication and addition.

Multiplies a vector with either the rotation matrix, the light matrix or the color matrix and then adds the translation vector or background color vector.

DCPL

Depth cue light color

First calculates a color from a light vector(normal vector of a plane multiplied with the light matrix and zero limited) and a provided RGB value. Then performs depth cueing by interpolating between the far color vector and the newfound color.

DPCS/DPCT

Depth cue single/triple

Performs depth cueing by interpolating between a color and the far color vector on one or three colors.

INTPL

Interpolation

Interpolates between a vector and the far color vector.

SQR

Square

Calculates the square of a vector.

NCS/NCT

Normal Color

Calculates a color from the normal of a point or plane and the light sources and colors. The basic color of the plane or point the normal refers to is assumed to be white.

NCDS/NCDT

Normal Color Depth Cue.

Same as NCS/NCT but also performs depth cueing (like DPCS/DPCT)

NCCS/NCCT

Same NCS/NCT, but the base color of the plane or point is taken into account.

CDP

A color is calculated from a light vector (base color is assumed to be white) and depth cueing is performed (like DPCS).

CC

A color is calculated from a light vector and a base color.

NCLIP

Calculates the outer product of three 2d points.(ie. 3 vertices which define a plane after projection.)

The 3 vertices should be stored clockwise according to the visual point:

```
  Z+
  /
 /____ X+
 |
 |
 Y+
```

If this is so, the result of this function will be negative if we are

facing the backside of the plane.

AVSZ3/AVSZ4

Adds 3 or 4 z values together and multplies them by a fixed point value. This value is normally chosen so that this function returns the average of the z values (usually further divided by 2 or 4 for easy adding to the OT)

OP

Calculates the outer product of 2 vectors.

GPF

Multiplies 2 vectors. Also returns the result as 24bit rgb value.

GPL

Multiplies a vector with a scalar and adds the result to another vector. Also returns the result as 24bit rgb value.

Instructions

The CPU has six special load and store instructions for the GTE registers, and an instruction to issue commands to the coprocessor.

rt CPU register 0-31

gd GTE data register 0-31

gc GTE control register 0-31

imm 16 bit immediate value

base CPU register 0-31

imm(base) address pointed to by base + imm.

b25 25 bit wide data field.

LWC2 gd, imm(base) stores value at imm(base) in GTE data register gd.

SWC2 gd, imm(base) stores GTE data register at imm(base).

MTC2 rt, gd stores register rt in GTE data register gd.

MFC2 rt, gd stores GTE data register gd in register rt.

CTC2 rt, gc stores register rt in GTE control register gc.

CFC2 rt, gc stores GTE control register in register rt.

COP2 b25 Issues a GTE command.

GTE load and store instructions have a delay of 2 instructions, for any GTE commands or operations accessing that register.

Programming the GTE.

Before use the GTE must be turned on. The GTE has bit 30 allocated to it in the status register of the system control coprocessor (cop0). Before any GTE instruction is used, this bit must be set.

GTE instructions and functions should not be used in

- Delay slots of jumps and branches

- Event handlers or interrupts.

If an instruction that reads a GTE register or a GTE command is executed before the current GTE command is finished, the CPU will hold until the instruction has finished. The number of cycles each GTE instruction takes is in the command list.

Registers.

The GTE has 32 data registers, and 32 control registers, each 32 bits wide. The control registers are commonly called Cop2C, while the data registers are called Cop2D. The following list describes their common use. The format is explained later on.

Control Registers (Cop2C)		
Number	**Name**	**Description**
0	**R11R12**	**Rotation matrix elements 1 to 1, 1 to 2**
1	**R13R21**	**Rotation matrix elements 1 to 3, 2 to 1**
2	**R22R23**	**Rotation matrix elements 2 to 2, 2 to 3**
3	**R31R32**	**Rotation matrix elements 3 to 1, 3 to 2**
4	**R33**	**Rotation matrix elements 3 to 3**
5	**TRX**	**Translation vector X**
6	**TRY**	**Translation vector Y**
7	**TRZ**	**Translation vector Z**
8	**L11L12**	**Light source matrix elements 1 to 1, 1 to 2**
9	**L13L21**	**Light source matrix elements 1 to 3, 2 to 1**
10	**L22L23**	**Light source matrix elements 2 to 2, 2 to 3**
11	**L31L32**	**Light source matrix elements 3 to 1, 3 to 2**
12	**L33**	**Light source matrix elements 3 to 3**

13	**RBK**	**Background color red component**
14	**BBK**	**Background color blue component**
15	**GBK**	**Background color green component**
16	**LR1LR2**	**Light color matrix source 1&2 red component**
17	**LR3LG1**	**Light color matrix source 3 red, 1 green component**
18	**LG2LG3**	**Light color matrix source 2&3 green component**
19	**LB1LB2**	**Light color matrix source 1&2 blue comp**
20	**LB3**	**Light color matrix source 3 blue component**
21	**RFC**	**Far color red component**
22	**GFC**	**Far color green component**
23	**BFC**	**Far color blue component**
24	**OFX**	**Screen offset X**
25	**OFY**	**Screen offset y**
26	**H**	**Projection plane distance**
27	**DQA**	**depth queuing parameter A.(coefficient.)**
28	**DQB**	**Depth queuing parameter B.(offset.)**
29	**ZSF3**	**Z3 average scale factor (normally 1/3)**
30	**ZSF4**	**Z4 average scale factor (normally 1/4)**
31	**FLAG**	**Returns any calculation errors.(See below)**

Control Register format

The GTE uses signed, fixed point registers for mathematics. The following is a bit-wise description of the registers.

R11R12					
31			0		
R11			R12		
Sign	integral part	fractional part	Sign	integral part	fractional part
1	3	12	1	3	12

R13R21					
31			0		
R13			R21		
Sign	integral part	fractional part	Sign	integral part	fractional part
1	3	12	1	3	12

R22R23					
31			0		
R22			R23		
Sign	integral part	fractional part	Sign	integral part	fractional part

1	3	12	1	3	12

R31R32					
31			0		
R31			R32		
Sign	integral part	fractional part	Sign	integral part	fractional part
1	3	12	1	3	12

R33					
31			0		
0			R33		
			Sign	integral part	fractional part
			1	3	12

TRX	
31	0
Sign	integral part
1	31

TRY	
31	0
Sign	integral part
1	31

TRZ	
31	0
Sign	integral part
1	31

L11L12					
31			0		
L11			L12		
Sign	integral part	fractional part	Sign	integral part	fractional part
1	3	12	1	3	12

L13L21					
31			0		
L13			L21		
Sign	integral part	fractional part	Sign	integral part	fractional part
1	3	12	1	3	12

L22L23					
31			0		
L22			L23		
Sign	integral part	fractional part	Sign	integral part	fractional part
1	3	12	1	3	12

L31L32					
31			0		
L31			L32		
Sign	integral part	fractional part	Sign	integral part	fractional part
1	3	12	1	3	12

L33			
31	0		
0	L33		
	Sign	integral part	fractional part
	1	3	12

RBK		
31		0
Sign	integral part	fractional part

1	19	12

GBK		
31		0
Sign	integral part	fractional part
1	19	12

BBK		
31		0
Sign	integral part	fractional part
1	19	12

LR1LR2					
31			0		
LR1			LR2		
Sign	integral part	fractional part	Sign	integral part	fractional part
1	3	12	1	3	12

LR3LLG1	
31	0
LR3	LG1

Sign	integral part	fractional part	Sign	integral part	fractional part
1	3	12	1	3	12

LG2LG3					
31			0		
LG2			LG3		
Sign	integral part	fractional part	Sign	integral part	fractional part
1	3	12	1	3	12

LB1LB2					
31			0		
LB1			LB2		
Sign	integral part	fractional part	Sign	integral part	fractional part
1	3	12	1	3	12

LB3			
31	0		
0	LB3		
	Sign	integral part	fractional part
	1	3	12

RFC		
31		**0**
Sign	**integral part**	**fractional part**
1	**27**	**4**

GFC		
31		**0**
Sign	**integral part**	**fractional part**
1	**27**	**4**

BFC		
31		**0**
Sign	**integral part**	**fractional part**
1	**27**	**4**

OFX		
31		**0**
Sign	**integral part**	**fractional part**
1	**15**	**16**

<table>
<tr><th colspan="3">0FY</th></tr>
<tr><td colspan="2">31</td><td>0</td></tr>
<tr><td>Sign</td><td>integral part</td><td>fractional part</td></tr>
<tr><td>1</td><td>15</td><td>16</td></tr>
</table>

<table>
<tr><th colspan="2">H</th></tr>
<tr><td>31</td><td>0</td></tr>
<tr><td rowspan="3">0</td><td>H</td></tr>
<tr><td>integral part</td></tr>
<tr><td>16</td></tr>
</table>

<table>
<tr><th colspan="4">DQA</th></tr>
<tr><td>31</td><td colspan="3">0</td></tr>
<tr><td rowspan="3">0</td><td colspan="3">DQA</td></tr>
<tr><td>Sign</td><td>integral part</td><td>fractional part</td></tr>
<tr><td>1</td><td>7</td><td>8</td></tr>
</table>

DQB

31	0		
0	DQB		
	Sign	integral part	fractional part
	1	7	8

ZF3			
31	0		
0	ZF3		
	Sign	integral part	fractional part
	1	3	12

DZF4			
31	0		
0	ZF4		
	Sign	integral part	fractional part
	1	3	12

FLAGS	
31	0

Flags bit description.

31 Logical sum of bits 30-23 and bits 18-13

30 Calculation test result #1 overflow (2^43 or more)

29 Calculation test result #2 overflow (2^43 or more)

28 Calculation test result #3 overflow (2^43 or more)

27 Calculation test result #1 underflow (less than -2^43)

26 Calculation test result #2 underflow (less than -2^43)

25 Calculation test result #3 underflow (less than -2^43)

24 Limiter A1 out of range (less than 0, or less than -2^15, or 2^15 or more)

23 Limiter A2 out of range (less than 0, or less than -2^15, or 2^15 or more)

22 Limiter A3 out of range (less than 0, or less than -2^15, or 2^15 or more)

21 Limiter B1 out of range (less than 0, or 2^8 or more)

20 Limiter B2 out of range (less than 0, or 2^8 or more)

19 Limiter B3 out of range (less than 0, or 2^8 or more)

18 Limiter C out of range (less than 0, or 2^16 or more)

17 Divide overflow generated (quotient of 2.0 or more)

16 Calculation test result #4 overflow (2^31 or more)

15 Calculation test result #4 underflow (less than -2^31)

14 Limiter D1 out of range (less than 2^10, or 2^10 or more)

13 Limiter D2 out of range (less than 2^10, or 2^10 or more)

12 Limiter E out of range (less than 0, or 2^12 or more)

Data Registers

Data registers consist of the other “half” of the GTE. Note in some functions format are different from the one that's given here. ***The numbers in the format fields are the signed, integer and fractional parts of the field. So 1,3,12 means signed(1 bit), 3 bits integral part, 12 bits fractional part.***

Data Registers (Cop2D)

Number	Name	r/w	31	16	15	0	Format	Description
0.	VXY0	r/w	VY0		VX0		1,3,12 or 1,15,0	Vector 0 X and Y
1	VZ0	r/w	0		VZ0		1,3,12 or 1,15,0	Vector 0 Z
2	VXY1	r/w	VY1		VX1		1,3,12 or 1,15,0	Vector 1 X and Y
3	VZ1	r/w	0		VZ1		1,3,12 or 1,15,0	Vector 1 Z
4	VXY2	r/w	VY2		VX2		1,3,12 or 1,15,0	Vector 2 X and Y
5	VZ2	r/w	0		VZ2		1,3,12 or 1,15,0	Vector 2 Z
6	RGB	r/w	Code, R		G,B		8 bits for each	RGB value. Code is passed, but not used in calculation
7	OTZ	r	0		OTZ		0,15,0	Z Average value.
8	IR0	r/w	Sign		IR0		1, 3,12	Intermediate value 0. Format may differ
9	IR1	r/w	Sign		IR1		1, 3,12	Intermediate value 1. Format may differ
10	IR2	r/w	Sign		IR2		1, 3,12	Intermediate value 2. Format may differ
11	IR3	r/w	Sign		IR3		1, 3,12	Intermediate value 3. Format may differ
12	SXY0	r/w	SX0		SY0		1,15, 0	Screen XY coordinate FIFO (Note 1)
13	SXY1	r/w	SX1		SY1		1,15, 0	Screen XY coordinate FIFO
14	SXY2	r/w	SX2		SY2		1,15, 0	Screen XY coordinate FIFO
15	SXYP	r/w	SXP		SYP		1,15, 0	Screen XY coordinate FIFO

16	SZ0	r/w	0	SZ0	0,16, 0	Screen Z FIFO (Note 1)
17	SZ1	r/w	0	SZ1	0,16, 0	Screen Z FIFO
18	SZ2	r/w	0	SZ2	0,16, 0	Screen Z FIFO
19	SZ3	r/w	0	SZ3	0,16, 0	Screen Z FIFO
20	RGB0	r/w	CD0,B0	G0,R0	8 bits each	Characteristic color FIFO(Note 1)
21	RGB1	r/w	CD1,B1	G1,R1	8 bits each	Characteristic color FIFO
22	RGB2	r/w	CD2,B2	G0,R2	8 bits each	CD2 is the bit pattern of currently executed function
23	RES1	-	-	-	-	Prohibited
24	MAC0	r/w	MAC0		1,31,0	Sum of products value 1
25	MAC1	r/w	MAC1		1,31,0	Sum of products value 1
26	MAC2	r/w	MAC2		1,31,0	Sum of products value 1
27	MAC3	r/w	MAC3		1,31,0	Sum of products value 1
28	IRGB	w	0	IB,IG,IR	Note 2	Note 2
29	ORGB	r	0	0B,0G,OR	Note 3	Note 3
30	LZCS	w	LZCS		1,31,0	Leading zero count source data (Note 4)
31	LZCR	r	LZCR		6,6,0	Leading zero count result (Note 4)

Note 1

The SXYx, SZx and RGBx are first in first out registers (FIFO). The last calculation result is stored in the last register, and previous results are stored in previous registers. So for example when a new SXY value is obtained the following happens:

SXY0 = SXY1

SXY1 = SXY2

SXY2 = SXYP

SXYP = result.

Note 2

IRGB							
0		R		G		B	
31	15	14	10	9	5	4	0

When writing a value to IRGB the following happens:

IR1 = IR format converted to (1,11,4)

IR2 = IG format converted to (1,11,4)

IR3 = IB format converted to (1,11,4)

Note 3

0RGB							
0		R		G		B	
31	15	14	10	9	5	4	0

When writing a value to IRGB the following happens:

IR = (IR1>>7) &0x1f

IG = (IR2>>7) &0x1f

IB = (IR3>>7) &0x1f

Note 4

Reading LZCR returns the leading 0 count of LZCS if LZCS is positive and the leading 1 count of LZCS if LZCS is negative.

GTE Commands.

This part describes the actual calculations performed by the various GTE functions. The first line contains the name of the function, the number of cycles it takes and a brief description. The second part contains any fields that may be set in the opcode and in the third line is the actual opcode. See the end of the list for the fields and their descriptions. Then follows a list of all registers which are needed in the calculation under the 'in', and a list of registers which modified under the 'out' with a brief description and the format of the data. Next follows the calculation which is performed after initiating the function. The format field left is the size in which the data is stored, the format field on the right contains the format in which the calculation is performed. At certain points in the calculation checks and limitations are done and their results stored in the flag register, see the table below. They are identified with the code from the second column of the table directly followed by square brackets enclosing the part of the calculation on which the check is performed. The additional Lm_ identifier means the value is limited to the bottom or ceiling of the check if it exceeds the boundary.

bit description

31 Checksum.

30 A1 Result larger than 43 bits and positive

29 A2 Result larger than 43 bits and positive

28 A3 Result larger than 43 bits and positive

27 A1 Result larger than 43 bits and negative

26 A2 Result larger than 43 bits and negative

25 A3 Result larger than 43 bits and negative

24 B1 Value negative(lm=1) or larger than 15 bits(lm=0)

23 B2 Value negative(lm=1) or larger than 15 bits(lm=0)

22 B3 Value negative(lm=1) or larger than 15 bits(lm=0)

21 C1 Value negative or larger than 8 bits.

20 C2 Value negative or larger than 8 bits.

19 C3 Value negative or larger than 8 bits.

18 D Value negative or larger than 16 bits.

17 E Divide overflow. (quotient > 2.0)

16 F Result larger than 31 bits and positive.

15 F Result larger than 31 bits and negative.

14 G1 Value larger than 10 bits.

13 G2 Value larger than 10 bits.

12 H Value negative or larger than 12 bits.

Name	Cycles	Command	Description
RTPS	15	cop2 0x0180001	Perspective transform

Fields: None

`In: V0 Vector to transform. [1,15,0]`

`R Rotation matrix [1,3,12]`

`TR Translation vector [1,31,0]`

`H View plane distance [0,16,0]`

`DQA Depth que interpolation values. [1,7,8]`

`DQB [1,7,8]`

`OFX Screen offset values. [1,15,16]`

`OFY [1,15,16]`

`Out: SXY fifo Screen XY coordinates.(short) [1,15,0]`

`SZ fifo Screen Z coordinate.(short) [0,16,0]`

`IR0 Interpolation value for depth queing. [1,3,12]`

`IR1 Screen X (short) [1,15,0]`

`IR2 Screen Y (short) [1,15,0]`

IR3 Screen Z (short) [1,15,0]

MAC1 Screen X (long) [1,31,0]

MAC2 Screen Y (long) [1,31,0]

MAC3 Screen Z (long) [1,31,0]

Calculation:

```
[1,31,0] MAC1=A1[TRX + R11*VX0 + R12*VY0 + R13*VZ0] [1,31,12]
[1,31,0] MAC2=A2[TRY + R21*VX0 + R22*VY0 + R23*VZ0] [1,31,12]
[1,31,0] MAC3=A3[TRZ + R31*VX0 + R32*VY0 + R33*VZ0] [1,31,12]
[1,15,0] IR1= Lm_B1[MAC1] [1,31,0]
[1,15,0] IR2= Lm_B2[MAC2] [1,31,0]
[1,15,0] IR3= Lm_B3[MAC3] [1,31,0]
SZ0<-SZ1<-SZ2<-SZ3
[0,16,0] SZ3= Lm_D(MAC3) [1,31,0]
SX0<-SX1<-SX2, SY0<-SY1<-SY2
[1,15,0] SX2= Lm_G1[F[OFX + IR1*(H/SZ)]] [1,27,16]
[1,15,0] SY2= Lm_G2[F[OFY + IR2*(H/SZ)]] [1,27,16]
[1,31,0] MAC0= F[DQB + DQA * (H/SZ)] [1,19,24]
[1,15,0] IR0= Lm_H[MAC0] [1,31,0]
```

Notes:

Z values are limited downwards at 0.5 * H. For smaller z values you'll have write your own routine.

Name	Cycles	Command	Description
RTPT	23	cop2 0x0280030	Perspective transform on 3 points

Fields: None

in: V0 Vector to transform. [1,15,0]

V1 [1,15,0]

V2 [1,15,0]

R Rotation matrix [1,3,12]

TR Translation vector [1,31,0]

H View plane distance [0,16,0]

DQA Depth que interpolation values. [1,7,8]

DQB [1,7,8]

OFX Screen offset values. [1,15,16]

OFY [1,15,16]

out: SXY fifo Screen XY coordinates.(short) [1,15,0]

SZ fifo Screen Z coordinate.(short) [0,16,0]

IR0 Interpolation value for depth queing. [1,3,12]

IR1 Screen X (short) [1,15,0]

IR2 Screen Y (short) [1,15,0]

IR3 Screen Z (short) [1,15,0]

MAC1 Screen X (long) [1,31,0]

MAC2 Screen Y (long) [1,31,0]

MAC3 Screen Z (long) [1,31,0]

Calculation: Same as RTPS, but repeats for V1 and V2.

Name	Cycles	Command	Description
MVMVA	8	cop2 0x0400012	Multiply vector by matrix and vector addition.

Fields: sf, mx, v, cv, lm

in: V0/V1/V2/IR Vector v0, v1, v2 or [IR1,IR2,IR3]

R/LLM/LCM Rotation, light or color matrix. [1,3,12]

TR/BK Translation or background color vector.

out: [IR1,IR2,IR3] Short vector

[MAC1,MAC2,MAC3] Long vector

Calculation:

MX = matrix specified by mx

V = vector specified by v

CV = vector specified by cv

MAC1=A1[CV1 + MX11*V1 + MX12*V2 + MX13*V3]

MAC2=A2[CV2 + MX21*V1 + MX22*V2 + MX23*V3]

MAC3=A3[CV3 + MX31*V1 + MX32*V2 + MX33*V3]

IR1=Lm_B1[MAC1]

IR2=Lm_B2[MAC2]

IR3=Lm_B3[MAC3]

Notes:

The cv field allows selection of the far color vector, but this vector

is not added correctly by the GTE.

Name	Cycles	Command	Description
DPCL	8	cop2 0x0680029	Depth Cue Color light

Fields:

In: RGB Primary color. R,G,B,CODE [0,8,0]

IR0 interpolation value. [1,3,12]

[IR1,IR2,IR3] Local color vector. [1,3,12]

CODE Code value from RGB. CODE [0,8,0]

FC Far color. [1,27,4]

Out: RGBn RGB fifo Rn,Gn,Bn,CDn [0,8,0]

[IR1,IR2,IR3] Color vector [1,11,4]

[MAC1,MAC2,MAC3] Color vector [1,27,4]

Calculation:

[1,27,4] MAC1=A1[R*IR1 + IR0*(Lm_B1[RFC - R * IR1])] [1,27,16]

[1,27,4] MAC2=A2[G*IR2 + IR0*(Lm_B1[GFC - G * IR2])] [1,27,16]

[1,27,4] MAC3=A3[B*IR3 + IR0*(Lm_B1[BFC - B * IR3])] [1,27,16]

[1,11,4] IR1=Lm_B1[MAC1] [1,27,4]

[1,11,4] IR2=Lm_B2[MAC2] [1,27,4]

[1,11,4] IR3=Lm_B3[MAC3] [1,27,4]

[0,8,0] Cd0<-Cd1<-Cd2<- CODE

[0,8,0] R0<-R1<-R2<- Lm_C1[MAC1] [1,27,4]

[0,8,0] G0<-G1<-G2<- Lm_C2[MAC2] [1,27,4]

[0,8,0] B0<-B1<-B2<- Lm_C3[MAC3] [1,27,4]

Name	Cycles	Command	Description
DPCS	**8**	**cop2 0x0780010**	**Depth Cueing**

Fields:

In: IR0 Interpolation value [1,3,12]

RGB Color R,G,B,CODE [0,8,0]

FC Far color RFC,GFC,BFC [1,27,4]

Out: RGBn RGB fifo Rn,Gn,Bn,CDn [0,8,0]

[IR1,IR2,IR3] Color vector [1,11,4]

[MAC1,MAC2,MAC3] Color vector [1,27,4]

Calculations:

[1,27,4] MAC1=A1[(R + IR0*(Lm_B1[RFC - R])] [1,27,16][lm=0]

[1,27,4] MAC2=A2[(G + IR0*(Lm_B1[GFC - G])] [1,27,16][lm=0]

[1,27,4] MAC3=A3[(B + IR0*(Lm_B1[BFC - B])] [1,27,16][lm=0]

[1,11,4] IR1=Lm_B1[MAC1] [1,27,4][lm=0]

[1,11,4] IR2=Lm_B2[MAC2] [1,27,4][lm=0]

[1,11,4] IR3=Lm_B3[MAC3] [1,27,4][lm=0]

[0,8,0] Cd0<-Cd1<-Cd2<- CODE

[0,8,0] R0<-R1<-R2<- Lm_C1[MAC1] [1,27,4]

[0,8,0] G0<-G1<-G2<- Lm_C2[MAC2] [1,27,4]

[0,8,0] B0<-B1<-B2<- Lm_C3[MAC3] [1,27,4]

Name	Cycles	Command	Description
DPCT	17	cop2 0x0F8002A	Depth cue color RGB0,RGB1,RGB2

Fields:

In: IR0 Interpolation value [1,3,12]

RGB0,RGB1,RGB2 Colors in RGB fifo. Rn,Gn,Bn,CDn [0,8,0]

FC Far color RFC,GFC,BFC [1,27,4]

Out: RGBn RGB fifo Rn,Gn,Bn,CDn [0,8,0]

[IR1,IR2,IR3] Color vector [1,11,4]

[MAC1,MAC2,MAC3] Color vector [1,27,4]

Calculations:

[1,27,4] MAC1=A1[R0+ IR0*(Lm_B1[RFC - R0])] [1,27,16][lm=0]

[1,27,4] MAC2=A2[G0+ IR0*(Lm_B1[GFC - G0])] [1,27,16][lm=0]

[1,27,4] MAC3=A3[B0+ IR0*(Lm_B1[BFC - B0])] [1,27,16][lm=0]

[1,11,4] IR1=Lm_B1[MAC1] [1,27,4][lm=0]

[1,11,4] IR2=Lm_B2[MAC2] [1,27,4][lm=0]

[1,11,4] IR3=Lm_B3[MAC3] [1,27,4][lm=0]

[0,8,0] Cd0<-Cd1<-Cd2<- CODE

[0,8,0] R0<-R1<-R2<- Lm_C1[MAC1] [1,27,4]

[0,8,0] G0<-G1<-G2<- Lm_C2[MAC2] [1,27,4]

[0,8,0] B0<-B1<-B2<- Lm_C3[MAC3] [1,27,4]

Performs this calculation 3 times, so all three RGB values have been replaced by the depth cued RGB values.

Name	Cycles	Command	Description
INTPL	**8**	**cop2 0x0980011**	**Interpolation of vector and far color**

Fields:

In: [IR1,IR2,IR3] Vector [1,3,12]

IR0 Interpolation value [1,3,12]

CODE Code value from RGB. CODE [0,8,0]

FC Far color RFC,GFC,BFC [1,27,4]

Out: RGBn RGB fifo Rn,Gn,Bn,CDn [0,8,0]

[IR1,IR2,IR3] Color vector [1,11,4]

[MAC1,MAC2,MAC3] Color vector [1,27,4]

Calculations:

[1,27,4] MAC1=A1[IR1 + IR0*(Lm_B1[RFC - IR1])] [1,27,16]

[1,27,4] MAC2=A2[IR2 + IR0*(Lm_B1[GFC - IR2])] [1,27,16]

[1,27,4] MAC3=A3[IR3 + IR0*(Lm_B1[BFC - IR3])] [1,27,16]

[1,11,4] IR1=Lm_B1[MAC1] [1,27,4]

[1,11,4] IR2=Lm_B2[MAC2] [1,27,4]

[1,11,4] IR3=Lm_B3[MAC3] [1,27,4]

[0,8,0] Cd0<-Cd1<-Cd2<- CODE

[0,8,0] R0<-R1<-R2<- Lm_C1[MAC1] [1,27,4]

[0,8,0] G0<-G1<-G2<- Lm_C2[MAC2] [1,27,4]

[0,8,0] B0<-B1<-B2<- Lm_C3[MAC3] [1,27,4]

Name	Cycles	Command	Description
SQR	5	cop2 0x0A00428	Square of vector

Fields: sf

in: [IR1,IR2,IR3] vector [1,15,0][1,3,12]

out: [IR1,IR2,IR3] vector^2 [1,15,0][1,3,12]

[MAC1,MAC2,MAC3] vector^2 [1,31,0][1,19,12]

Calculation: (left format sf=0, right format sf=1)

[1,31,0][1,19,12] MAC1=A1[IR1*IR1] [1,43,0][1,31,12]

[1,31,0][1,19,12] MAC2=A2[IR2*IR2] [1,43,0][1,31,12]

[1,31,0][1,19,12] MAC3=A3[IR3*IR3] [1,43,0][1,31,12]

[1,15,0][1,3,12] IR1=Lm_B1[MAC1] [1,31,0][1,19,12][lm=1]

[1,15,0][1,3,12] IR2=Lm_B2[MAC2] [1,31,0][1,19,12][lm=1]

[1,15,0][1,3,12] IR3=Lm_B3[MAC3] [1,31,0][1,19,12][lm=1]

Name	Cycles	Command	Description

NCS	14	cop2 0x0C8041E	Normal color v0

Fields:

In: V0 Normal vector [1,3,12]

BK Background color RBK,GBK,BBK [1,19,12]

CODE Code value from RGB. CODE [0,8,0]

LCM Color matrix [1,3,12]

LLM Light matrix [1,3,12]

Out: RGBn RGB fifo. Rn,Gn,Bn,CDn [0,8,0]

[IR1,IR2,IR3] Color vector [1,11,4]

[MAC1,MAC2,MAC3] Color vector [1,27,4]

```
[1,19,12] MAC1=A1[L11*VX0 + L12*VY0 + L13*VZ0] [1,19,24]

[1,19,12] MAC2=A2[L21*VX0 + L22*VY0 + L23*VZ0] [1,19,24]

[1,19,12] MAC3=A3[L31*VX0 + L32*VY0 + L33*VZ0] [1,19,24]

[1,3,12] IR1= Lm_B1[MAC1] [1,19,12][lm=1]

[1,3,12] IR2= Lm_B2[MAC2] [1,19,12][lm=1]

[1,3,12] IR3= Lm_B3[MAC3] [1,19,12][lm=1]

[1,19,12] MAC1=A1[RBK + LR1*IR1 + LR2*IR2 + LR3*IR3] [1,19,24]

[1,19,12] MAC2=A2[GBK + LG1*IR1 + LG2*IR2 + LG3*IR3] [1,19,24]

[1,19,12] MAC3=A3[BBK + LB1*IR1 + LB2*IR2 + LB3*IR3] [1,19,24]

[1,3,12] IR1= Lm_B1[MAC1] [1,19,12][lm=1]

[1,3,12] IR2= Lm_B2[MAC2] [1,19,12][lm=1]

[1,3,12] IR3= Lm_B3[MAC3] [1,19,12][lm=1]

[0,8,0] Cd0<-Cd1<-Cd2<- CODE

[0,8,0] R0<-R1<-R2<- Lm_C1[MAC1] [1,27,4]

[0,8,0] G0<-G1<-G2<- Lm_C2[MAC2] [1,27,4]

[0,8,0] B0<-B1<-B2<- Lm_C3[MAC3] [1,27,4]
```

Name	Cycles	Command	Description
NCT	30	cop2 0x0D80420	Normal color v0, v1, v2

Fields:

In: V0,V1,V2 Normal vector [1,3,12]

BK Background color RBK,GBK,BBK [1,19,12]

CODE Code value from RGB. CODE [0,8,0]

LCM Color matrix [1,3,12]

LLM Light matrix [1,3,12]

Out: RGBn RGB fifo. Rn,Gn,Bn,CDn [0,8,0]

[IR1,IR2,IR3] Color vector [1,11,4]

[MAC1,MAC2,MAC3] Color vector [1,27,4]

Calculation: Same as NCS, but repeated for V1 and V2.

Name	Cycles	Command	Description
NCDS	19	cop2 0x0E80413	Normal color depth cuev0

Fields:

In: V0 Normal vector [1,3,12]

BK Background color RBK,GBK,BBK [1,19,12]

RGB Primary color R,G,B,CODE [0,8,0]

LLM Light matrix [1,3,12]

LCM Color matrix [1,3,12]

IR0 Interpolation value [1,3,12]

Out: RGBn RGB fifo. Rn,Gn,Bn,CDn [0,8,0]

[IR1,IR2,IR3] Color vector [1,11,4]

[MAC1,MAC2,MAC3] Color vector [1,27,4]

Calculation:

[1,19,12] MAC1=A1[L11*VX0 + L12*VY0 + L13*VZ0] [1,19,24]

[1,19,12] MAC2=A1[L21*VX0 + L22*VY0 + L23*VZ0] [1,19,24]

[1,19,12] MAC3=A1[L31*VX0 + L32*VY0 + L33*VZ0] [1,19,24]

[1,3,12] IR1= Lm_B1[MAC1] [1,19,12][lm=1]

[1,3,12] IR2= Lm_B2[MAC2] [1,19,12][lm=1]

[1,3,12] IR3= Lm_B3[MAC3] [1,19,12][lm=1]

[1,19,12] MAC1=A1[RBK + LR1*IR1 + LR2*IR2 + LR3*IR3] [1,19,24]

[1,19,12] MAC2=A1[GBK + LG1*IR1 + LG2*IR2 + LG3*IR3] [1,19,24]

[1,19,12] MAC3=A1[BBK + LB1*IR1 + LB2*IR2 + LB3*IR3] [1,19,24]

[1,3,12] IR1= Lm_B1[MAC1] [1,19,12][lm=1]

[1,3,12] IR2= Lm_B2[MAC2] [1,19,12][lm=1]

[1,3,12] IR3= Lm_B3[MAC3] [1,19,12][lm=1]

[1,27,4] MAC1=A1[R*IR1 + IR0*(Lm_B1[RFC-R*IR1])] [1,27,16][lm=0]

[1,27,4] MAC2=A1[G*IR2 + IR0*(Lm_B2[GFC-G*IR2])] [1,27,16][lm=0]

[1,27,4] MAC3=A1[B*IR3 + IR0*(Lm_B3[BFC-B*IR3])] [1,27,16][lm=0]

[1,3,12] IR1= Lm_B1[MAC1] [1,27,4][lm=1]

[1,3,12] IR2= Lm_B2[MAC2] [1,27,4][lm=1]

[1,3,12] IR3= Lm_B3[MAC3] [1,27,4][lm=1]

[0,8,0] Cd0<-Cd1<-Cd2<- CODE

[0,8,0] R0<-R1<-R2<- Lm_C1[MAC1] [1,27,4]

[0,8,0] G0<-G1<-G2<- Lm_C2[MAC2] [1,27,4]

[0,8,0] B0<-B1<-B2<- Lm_C3[MAC3] [1,27,4]

Name	Cycles	Command	Description
NCDT	**44**	**cop2 0x0F80416**	**Normal color depth cue v0, v1, v2**

Fields:

In: V0 Normal vector [1,3,12]

V1 Normal vector [1,3,12]

V2 Normal vector [1,3,12]

BK Background color RBK,GBK,BBK [1,19,12]

FC Far color RFC,GFC,BFC [1,27,4]

RGB Primary color R,G,B,CODE [0,8,0]

LLM Light matrix [1,3,12]

LCM Color matrix [1,3,12]

IR0 Interpolation value [1,3,12]

Out: RGBn RGB fifo. Rn,Gn,Bn,CDn [0,8,0]

[IR1,IR2,IR3] Color vector [1,11,4]

[MAC1,MAC2,MAC3] Color vector [1,27,4]

Calculation:

Same as NCDS but repeats for v1 and v2.

Name	Cycles	Command	Description
NCCS	17	cop2 0x108041B	Normal color col. v0

Fields:

In: V0 Normal vector [1,3,12]

BK Background color RBK,GBK,BBK [1,19,12]

RGB Primary color R,G,B,CODE [0,8,0]

LLM Light matrix [1,3,12]

LCM Color matrix [1,3,12]

Out: RGBn RGB fifo. Rn,Gn,Bn,CDn [0,8,0]

[IR1,IR2,IR3] Color vector [1,11,4]

[MAC1,MAC2,MAC3] Color vector [1,27,4]

Calculation:

[1,19,12] MAC1=A1[L11*VX0 + L12*VY0 + L13*VZ0] [1,19,24]

[1,19,12] MAC2=A2[L21*VX0 + L22*VY0 + L23*VZ0] [1,19,24]

[1,19,12] MAC3=A3[L31*VX0 + L32*VY0 + L33*VZ0] [1,19,24]

[1,3,12] IR1= Lm_B1[MAC1] [1,19,12][lm=1]

[1,3,12] IR2= Lm_B2[MAC2] [1,19,12][lm=1]

[1,3,12] IR3= Lm_B3[MAC3] [1,19,12][lm=1]

[1,19,12] MAC1=A1[RBK + LR1*IR1 + LR2*IR2 + LR3*IR3] [1,19,24]

[1,19,12] MAC2=A2[GBK + LG1*IR1 + LG2*IR2 + LG3*IR3] [1,19,24]

[1,19,12] MAC3=A3[BBK + LB1*IR1 + LB2*IR2 + LB3*IR3] [1,19,24]

[1,3,12] IR1= Lm_B1[MAC1] [1,19,12][lm=1]

[1,3,12] IR2= Lm_B2[MAC2] [1,19,12][lm=1]

[1,3,12] IR3= Lm_B3[MAC3] [1,19,12][lm=1]

[1,27,4] MAC1=A1[R*IR1] [1,27,16]

[1,27,4] MAC2=A2[G*IR2] [1,27,16]

[1,27,4] MAC3=A3[B*IR3] [1,27,16]

[1,3,12] IR1= Lm_B1[MAC1] [1,27,4][lm=1]

[1,3,12] IR2= Lm_B2[MAC2] [1,27,4][lm=1]

[1,3,12] IR3= Lm_B3[MAC3] [1,27,4][lm=1]

[0,8,0] Cd0<-Cd1<-Cd2<- CODE

[0,8,0] R0<-R1<-R2<- Lm_C1[MAC1] [1,27,4]

[0,8,0] G0<-G1<-G2<- Lm_C2[MAC2] [1,27,4]

[0,8,0] B0<-B1<-B2<- Lm_C3[MAC3] [1,27,4]

Name	Cycles	Command	Description
NCCT	39	cop2 0x118043F	Normal color col.v0, v1, v2

Fields:

In: V0 Normal vector 1 [1,3,12]

V1 Normal vector 2 [1,3,12]

V2 Normal vector 3 [1,3,12]

BK Background color RBK,GBK,BBK [1,19,12]

RGB Primary color R,G,B,CODE [0,8,0]

LLM Light matrix [1,3,12]

LCM Color matrix [1,3,12]

Out: RGBn RGB fifo. Rn,Gn,Bn,CDn [0,8,0]

[IR1,IR2,IR3] Color vector [1,11,4]

[MAC1,MAC2,MAC3] Color vector [1,27,4]

Calculation:

Same as NCCS but repeats for v1 and v2.

Name	Cycles	Command	Description
CDP	13	cop2 0x1280414	Color Depth Queue

Fields:

In: [IR1,IR2,IR3] Vector [1,3,12]

RGB Primary color R,G,B,CODE [0,8,0]

IR0 Interpolation value [1,3,12]

BK Background color RBK,GBK,BBK [1,19,12]

LCM Color matrix [1,3,12]

FC Far color RFC,GFC,BFC [1,27,4]

Out: RGBn RGB fifo Rn,Gn,Bn,CDn [0,8,0]

[IR1,IR2,IR3] Color vector [1,11,4]

[MAC1,MAC2,MAC3] Color vector [1,27,4]

Calculation:

[1,19,12] MAC1=A1[RBK + LR1*IR1 + LR2*IR2 + LR3*IR3] [1,19,24]

[1,19,12] MAC2=A2[GBK + LG1*IR1 + LG2*IR2 + LG3*IR3] [1,19,24]

[1,19,12] MAC3=A3[BBK + LB1*IR1 + LB2*IR2 + LB3*IR3] [1,19,24]

[1,3,12] IR1= Lm_B1[MAC1] [1,19,12][lm=1]

[1,3,12] IR2= Lm_B2[MAC2] [1,19,12][lm=1]

[1,3,12] IR3= Lm_B3[MAC3] [1,19,12][lm=1]

[1,27,4] MAC1=A1[R*IR1 + IR0*(Lm_B1[RFC-R*IR1])] [1,27,16][lm=0]

[1,27,4] MAC2=A2[G*IR2 + IR0*(Lm_B2[GFC-G*IR2])] [1,27,16][lm=0]

[1,27,4] MAC3=A3[B*IR3 + IR0*(Lm_B3[BFC-B*IR3])] [1,27,16][lm=0]

[1,3,12] IR1= Lm_B1[MAC1] [1,27,4][lm=1]

[1,3,12] IR2= Lm_B2[MAC2] [1,27,4][lm=1]

[1,3,12] IR3= Lm_B3[MAC3] [1,27,4][lm=1]

[0,8,0] Cd0<-Cd1<-Cd2<- CODE

[0,8,0] R0<-R1<-R2<- Lm_C1[MAC1] [1,27,4]

[0,8,0] G0<-G1<-G2<- Lm_C2[MAC2] [1,27,4]

[0,8,0] B0<-B1<-B2<- Lm_C3[MAC3] [1,27,4]

Name	Cycles	Command	Description
CC	11	cop2 0x138041C	Color Col.

Fields:

In: [IR1,IR2,IR3] Vector [1,3,12]

BK Background color RBK,GBK,BBK [1,19,12]

RGB Primary color R,G,B,CODE [0,8,0]

LCM Color matrix [1,3,12]

Out: RGBn RGB fifo. Rn,Gn,Bn,CDn [0,8,0]

[IR1,IR2,IR3] Color vector [1,11,4]

[MAC1,MAC2,MAC3] Color vector [1,27,4]

Calculations:

[1,19,12] MAC1=A1[RBK + LR1*IR1 + LR2*IR2 + LR3*IR3] [1,19,24]

[1,19,12] MAC2=A2[GBK + LG1*IR1 + LG2*IR2 + LG3*IR3] [1,19,24]

[1,19,12] MAC3=A3[BBK + LB1*IR1 + LB2*IR2 + LB3*IR3] [1,19,24]

[1,3,12] IR1= Lm_B1[MAC1] [1,19,12][lm=1]

[1,3,12] IR2= Lm_B2[MAC2] [1,19,12][lm=1]

[1,3,12] IR3= Lm_B3[MAC3] [1,19,12][lm=1]

[1,27,4] MAC1=A1[R*IR1] [1,27,16]

[1,27,4] MAC2=A2[G*IR2] [1,27,16]

[1,27,4] MAC3=A3[B*IR3] [1,27,16]

[1,3,12] IR1= Lm_B1[MAC1] [1,27,4][lm=1]

[1,3,12] IR2= Lm_B2[MAC2] [1,27,4][lm=1]

[1,3,12] IR3= Lm_B3[MAC3] [1,27,4][lm=1]

[0,8,0] Cd0<-Cd1<-Cd2<- CODE

[0,8,0] R0<-R1<-R2<- Lm_C1[MAC1] [1,27,4]

[0,8,0] G0<-G1<-G2<- Lm_C2[MAC2] [1,27,4]

[0,8,0] B0<-B1<-B2<- Lm_C3[MAC3] [1,27,4]

-

Name	Cycles	Command	Description
NCLIP	8	cop2 0x1400006	Normal clipping

Fields:

in: SXY0,SXY1,SXY2 Screen coordinates [1,15,0]

out: MAC0 Outerproduct of SXY1 and SXY2 with [1,31,0]

SXY0 as origin.

Calculation:

[1,31,0] MAC0 = F[SX0*SY1+SX1*SY2+SX2*SY0-SX0*SY2-SX1*SY0-SX2*SY1] [1,43,0]

Name	Cycles	Command	Description
AVSZ3	**5**	**cop2 0x158002D**	**Average of three Z values**

Fields:

in: SZ1, SZ2, SZ3 Z-Values [0,16,0]

ZSF3 Divider [1,3,12]

out: OTZ Average. [0,16,0]

MAC0 Average. [1,31,0]

Calculation:

[1,31,0] MAC0=F[ZSF3*SZ1 + ZSF3*SZ2 + ZSF3*SZ3] [1,31,12]

[0,16,0] OTZ=Lm_D[MAC0] [1,31,0]

Name	Cycles	Command	Description
AVSZ4	**6**	**cop2 0x168002E**	**Average of four Z values**

Fields:

in: SZ1,SZ2,SZ3,SZ4 Z-Values [0,16,0]

ZSF4 Divider [1,3,12]

out: OTZ Average. [0,16,0]

MAC0 Average. [1,31,0]

Calculation:

[1,31,0] MAC0=F[ZSF4*SZ0 + ZSF4*SZ1 + ZSF4*SZ2 + ZSF4*SZ3] [1,31,12]

[0,16,0] OTZ=Lm_D[MAC0] [1,31,0]

Name	Cycles	Command	Description
OP	6	cop2 0x170000C	Outer Product

Fields: sf

in: [R11R12,R22R23,R33] vector 1

[IR1,IR2,IR3] vector 2

out: [IR1,IR2,IR3] outer product

[MAC1,MAC2,MAC3] outer product

Calculation: (D1=R11R12,D2=R22R23,D3=R33)

MAC1=A1[D2*IR3 - D3*IR2]

MAC2=A2[D3*IR1 - D1*IR3]

MAC3=A3[D1*IR2 - D2*IR1]

IR1=Lm_B1[MAC0]

IR2=Lm_B2[MAC1]

IR3=Lm_B3[MAC2]

Name	Cycles	Command	Description
GPF	6	cop2 0x190003D	General purpose interpolation

Fields: sf

in: IR0 scaling factor

CODE code field of RGB

[IR1,IR2,IR3] vector

out: [IR1,IR2,IR3] vector

[MAC1,MAC2,MAC3] vector

RGB2 RGB fifo.

Calculation:

MAC1=A1[IR0 * IR1]

MAC2=A2[IR0 * IR2]

MAC3=A3[IR0 * IR3]

IR1=Lm_B1[MAC1]

IR2=Lm_B2[MAC2]

IR3=Lm_B3[MAC3]

[0,8,0] Cd0<-Cd1<-Cd2<- CODE

[0,8,0] R0<-R1<-R2<- Lm_C1[MAC1]

[0,8,0] G0<-G1<-G2<- Lm_C2[MAC2]

[0,8,0] B0<-B1<-B2<- Lm_C3[MAC3]

Name	Cycles	Command	Description
GPL	5	cop2 0x1A0003E	general purpose interpolation

Fields: sf

in: IR0 scaling factor

CODE code field of RGB

[IR1,IR2,IR3] vector

[MAC1,MAC2,MAC3] vector

out: [IR1,IR2,IR3] vector

[MAC1,MAC2,MAC3] vector

RGB2 RGB fifo.

Calculation:

```
MAC1=A1[MAC1 + IR0 * IR1]
MAC2=A2[MAC2 + IR0 * IR2]
MAC3=A3[MAC3 + IR0 * IR3]
IR1=Lm_B1[MAC1]
IR2=Lm_B2[MAC2]
IR3=Lm_B3[MAC3]
[0,8,0] Cd0<-Cd1<-Cd2<- CODE
[0,8,0] R0<-R1<-R2<- Lm_C1[MAC1]
[0,8,0] G0<-G1<-G2<- Lm_C2[MAC2]
[0,8,0] B0<-B1<-B2<- Lm_C3[MAC3]
```

- **Field descriptions.**

24	20	19	18	17	16	15	14	13	12	11	10	9	0
		sf	**mx**		**v**		**cv**				**lm**		

sf 0 vector format (1,31, 0)

1 vector format (1,19,12)

mx 0 Multiply with rotation matrix

1 Multiply with light matrix

2 Multiply with color matrix

3 Unknown

v 0 V0 source vector (short)

1 V1 source vector (short)

2 V2 source vector (short)

3 IR source vector (long)

cv 0 Add translation vector

1 Add back color vector

2 Unknown

3 Add no vector

lm 0 No negative limit.

1 Limit negative results to 0.

A list of common MVMVA instructions:

Name	Cycles	Command	Description

rtv0	**-**	**cop2 0x0486012**	**v0 * rotmatrix**
rtv1	**-**	**cop2 0x048E012**	**v1 * rotmatrix**
rtv2	**-**	**cop2 0x0496012**	**v2 * rotmatrix**
rtir12	**-**	**cop2 0x049E012**	**ir * rotmatrix**
rtir0	**-**	**cop2 0x041E012**	**ir * rotmatrix**
rtv0tr	**-**	**cop2 0x0480012**	**v0 * rotmatrix + tr vector**
rtv1tr	**-**	**cop2 0x0488012**	**v1 * rotmatrix + tr vector**
rtv2tr	**-**	**cop2 0x0490012**	**v2 * rotmatrix + tr vector**
rtirtr	**-**	**cop2 0x0498012**	**ir * rotmatrix + tr vector**
rtv0bk	**-**	**cop2 0x0482012**	**v0 * rotmatrix + bk vector**
rtv1bk	**-**	**cop2 0x048A012**	**v1 * rotmatrix + bk vector**
rtv2bk	**-**	**cop2 0x0492012**	**v2 * rotmatrix + bk vector**
rtirbk	**-**	**cop2 0x049A012**	**ir * rotmatrix + bk vector**
ll	**-**	**cop2 0x04A6412**	**v0 * light matrix. Lower limit result to 0**
llv0	**-**	**cop2 0x04A6012**	**v0 * light matrix**
llv1	**-**	**cop2 0x04AE012**	**v1 * light matrix**

llv2	**-**	**cop2 0x04B6012**	**v2 * light matrix**
llvir	**-**	**cop2 0x04BE012**	**ir * light matrix**
llv0tr	**-**	**cop2 0x04A0012**	**v0 * light matrix + tr vector**
llv1tr	**-**	**cop2 0x04A8012**	**v1 * light matrix + tr vector**
llv2tr	**-**	**cop2 0x04B0012**	**v2 * light matrix + tr vector**
llirtr	**-**	**cop2 0x04B8012**	**ir * light matrix + tr vector**
llv0bk	**-**	**cop2 0x04A2012**	**v0 * light matrix + bk vector**
llv1bk	**-**	**cop2 0x04AA012**	**v1 * light matrix + bk vector**
llv2bk	**-**	**cop2 0x04B2012**	**v2 * light matrix + bk vector**
llirbk	**-**	**cop2 0x04BA012**	**ir * light matrix + bk vector**
lc	**-**	**cop2 0x04DA412**	**v0 * color matrix, Lower limit clamped to 0**
lcv0	**-**	**cop2 0x04C6012**	**v0 * color matrix**
lcv1	**-**	**cop2 0x04CE012**	**v1 * color matrix**
lcv2	**-**	**cop2 0x04D6012**	**v2 * color matrix**
lcvir	**-**	**cop2 0x04DE012**	**ir * color matrix**
lcv0tr	**-**	**cop2 0x04C0012**	**v0 * color matrix + tr vector**

lcv1tr	**-**	**cop2 0x04C8012**	**v1 * color matrix + tr vector**
lcv2tr	**-**	**cop2 0x04D0012**	**v2 * color matrix + tr vector**
lcirtr	**-**	**cop2 0x04D8012**	**ir * color matrix + tr vector**
lev0bk	**-**	**cop2 0x04C2012**	**v0 * color matrix + bk vector**
lev1bk	**-**	**cop2 0x04CA012**	**v1 * color matrix + bk vector**
lev2bk	**-**	**cop2 0x04D2012**	**v2 * color matrix + bk vector**
leirbk	**-**	**cop2 0x04DA012**	**ir * color matrix + bk vector**

- **Other instructions:**

Name	Cycles	Command	Description	Format
sqr12	**-**	**cop2 0x0A80428**	**square of ir**	**1,19,12**
sqr0	**-**	**cop2 0x0A80428**	**square of ir**	**1,31, 0**
op12	**-**	**cop2 0x178000C**	**outer product**	**1,19,12**
op0	**-**	**cop2 0x170000C**	**outer product**	**1,31, 0**
gpf12	**-**	**cop2 0x198003D**	**general purpose interpolation**	**1,19,12**
gpf0	**-**	**cop2 0x190003D**	**general purpose interpolation**	**1,31, 0**

gpl12	**-**	**cop2 0x1A8003E**	**general purpose interpolation**	**1,19,12**
gpl0	**-**	**cop2 0x1A0003E**	**general purpose interpolation**	**1,31, 0**

The Motion Decoder (MDEC)

The Motion Decoder (MDEC) is a special controller chip that takes a compressed JPEG-like images and decompresses them into 24-bit bitmapped images for display by the GPU. The MDEC can only decompress a 16x16 pixel 24-bit image at at time,called "Macroblocks" These Macrobock are encoded block that uses the YUV (YCbCr) color scheme with Discrete Cosine Transformation (DCT) and Run Length Encoding (RLE) applied The MDEC also performs 24 to 16 bit color conversion to prepare it for whatever color depth the GPU is in. Due to the extremely high speed that the decompression is done, the decompressed RGB bitmaps can be combined to from larger pictures and then ,if displayed in sequential order, to produce movies. The maximum speed is about 9,000 macroblocks per second, thereby making a movie that is 320x240 able to be played at about 30 frames per second. MDEC data can only be sent/received via DMA channels 0 and 1. DMA channel 0 is for uncompressed data going in and channel 1 is for retrieval of the uncompressed macroblocks. The MDEC gets controlled via the MDEC control register at location $1f80_1820. The current status of the MDEC can be checked using the MDEC status register at $1f80_1824. The following is a layout of the registers.

$1f80_1820 (mdec0)

write:

31	28	27	26	25	24	0
u		RGB24	u	STP	u	

Note: The first word of every data segment in a str-file is a control word written to this register.

u Unknown

RGB24 should be set to 0 for 24-bit color and to 1 for 16-bit. In 16-bit mode

STP toggles whether to set bit 15 of the decompressed data (semi-transparency)

$1f80_1824 (mdec1)

read:

31	30	29	28	27	26	25	24	23	22	0
FIFO		InSync	DREQ		u	RGB24	OutSync	STP	u	

u Unknown

FIFO First-In-First-Out buffer state

InSync MDEC is busy decompressing data

OutSync MDEC is trasnferring data to man memory

DREQ Data Request

RGB24 0 for 24-bit color and to 1 for 16-bit. In 16-bit mode

STP toggles whether to set bit 15 of the decompressed data (semi-transparency)

write:

31	30	0
reset	u	

u Unknown

reset reset MDEC

MDEC Data Fomat

The MDEC uses a 'lossy' picture format simalar to that of the JPEG file format. A typical picture, before being put into the MDEC via DMA, is of the following format;

header
macroblock
...
macroblock
footer

- **The header is a 32 byte word.**

31	**16**	**15**	**0**
0x3800		**sixe**	

0x3800 Data ID

size size if data after the header

- **The Macrobocks are further broken up as follows**

Cb block
Cr block
Y0 block
Y1 block
Y2 block
Y3 block

Cb,Cr The color difference blocks

Y0,Y1,Y2,Y3 The Luminescence blocks

- **Within each block the DCT informaton and RLE compressed is is stored.**

15	**0**
DCT	
RLE	
...	
RLE	
EOD	

- **DCT DCT data, it has the quantization factor and the Direct Current (DC) reference**

15	**10**	**9**	**0**

Q	DC

Q Quantization factor (6 bits, unsigned)

DC Direct Current reference (10 bits, signed)

- **RLE Run length data**

15	10	9	0
LENGTH		DATA	

LENGTH The number if zeros between data (6 bits, unsigned)

DATA The data (10 bits, signed)

- **EOD End Of Data(Footer)**

15	0
0xfe00	

Lets the MDEC know a block is done. The footer is also the same thing.

SOUND

SPU - Sound Processing Unit

Introduction.

The SPU is the unit responsible for all aural capabilities of the psx. It handles 24 voices, has a 512kb sound buffer. It also has ADSR envelope filters for each voice and lots of other features.

The Sound Buffer

The SPU has control over a 512kb sound buffer. Data is stored compressed into blocks of 16 bytes. Each block contains 14 packed sample bytes and two header bytes, one for the packing and one for sample end and looping information. One such block is decoded into 28 sample bytes (= 14 16bit samples).

In the first 4 kb of the buffer the SPU stores the decoded data of CD audio after volume processing and the sound data of voice 1 and voice 3 after envelope processing. The decoded data is stored as 16 bit signed values,

one sample per clock (44.1 khz).

Following this first 4kb are 8 bytes reserved by the system. The memory beyond that is free to store samples, up to the reverb work area if the effect processor is used. The size of this work area depends on which type of effect is being processed. More on that later.

Memory layout	
0x00000-0x003ff	CD audio left
0x00400-0x007ff	CD audio right
0x00800-0x00bff	Voice 1
0x00c00-0x00fff	Voice 3
0x01000-0x0100f	System area.
0x01008-0xxxxxx	Sound data area.

0x0xxxx-0x7ffff	**Reverb work area.**

Voices

The SPU has 24 hardware voices. These voices can be used to reproduce sample data, noise or can be used as frequency modulator on the next voice. Each voice has it's own programmable ADSR envelope filter. The main volume can be programmed independently for left and right output.

The ADSR envelope filter works as follows:

Ar Attack rate, which specifies the speed at which the volume increases from zero to it's maximum value, as soon as the note on is given. The slope can be set to lineair or exponential.

Dr Decay rate specifies the speed at which the volume decreases to the sustain level. Decay is always decreasing exponentially.

Sl Sustain level, base level from which sustain starts.

Sr Sustain rate is the rate at which the volume of the sustained note increases or decreases. This can be either lineair or exponential.

Rr Release rate is the rate at which the volume of the note decreases as soon as the note off is given.

lvl Volume level

t Time

The overal volume can also be set to sweep up or down lineairly or exponentially from it's current value. This can be done seperately for left and right.

SPU Operation

The SPU occupies the area 0x1f80_1c00-0x1f80_1dff. All registers are 16 bit wide.

0x1f80_1c00-0x1f80_1d7f Voice data area. For each voice there are 8 16 bit registers structured like this:

0x1f80_1xx0-0x1f80_1xx2 Volume

(*xx* = 0xc0 + voice number)	
0x1f80_1xx0	Volume Left
0x1f80_1xx2	Volume Right

Volume mode:

15	14	13	0
0	S	VV	

VV 0x0000-0x3fff Voice volume.

S 0 Phase Normal

1 Inverted

Sweep mode:

15	14	13	12	11	7	6	0
1	Sl	Dr	Ph			VV	

VV 0x0000-0x007f Voice volume.

Sl 0 Lineair slope

1 Exponential slope

Dr 0 Increase

1 Decrease

Ph 0 Normal phase

1 Inverted phase

In sweep mode, the current volume increases to its maximum value, or decreases to its mimimum value, according to mode. Choose phase equal to the the phase of the current volume.

0x1f80_1xx4 Pitch

15	14	13	0
		Pt	

Pt 0x0000-0x3fff Specifies pitch.

Any value can be set, table shows only octaves:

0x0200 -3 octaves

0x0400 -2

0x0800 -1

0x1000 sample pitch

0x2000 +1

0x3fff +2

0x1f80_1xx6 Start address of Sound

15	0
Addr	

Addr Startaddress of sound in Sound buffer /8

0x1f80_1xx8 Attack/Decay/Sustain level

15	14	8	7	4	3	0
Am	Ar		Dr		Sl	

Am 0 Attack mode Linear

1 Exponential

Ar 0-7f attack rate

Dr 0-f decay rate

Sl 0-f sustain level

0x1f80_1xxa Sustain rate, Release Rate.

15	14	13	12	6	5	4	0
Sm	Sd	0	Sr		Rm	Rr	

Sm 0 sustain rate mode linear

1 exponential

Sd 0 sustain rate mode increase

1 decrease

Sr 0-7f Sustain Rate

Rm 0 Linear decrease

1 Exponential decrease

Rr 0-1f Release Rate

Note: decay mode is always Expontial decrease, and thus cannot be set.

0x1f80_1xxc Current ADSR volume

15	0
ASDRvol	

ADSRvol Returns the current envelope volume when read.

0x1f80_1xxe Repeat address.

15	0
Ra	

Ra 0x0000-0xffff Address sample loops to at end.

Note: Setting this register only has effect after the voice has started (ie. KeyON), else the loop address gets reset by the sample.

SPU Global Registers

0x1f801d80 Main volume left

0x1f801d82 Main volume right

15	0
MVol	

Mvol 0x0000-0xffff Main volume

Sets Main volume, these work the same as the channel volume registers. See those for details.

0x1f801d84 Reverberation depth left

0x1f801d86 Reverberation depth right

15	14	0
P	Rvd	

Rvd 0x0000-0x7fff Sets the wet volume for the effect.

P 0 Normal phase

1 Inverted phase

Following registers have a common layout:

first register:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
c15	c14	c13	c12	c11	c10	c9	c8	c7	c6	c5	c4	c3	c2	c1	c0

second register:

15	8	7	6	5	4	3	2	1	0
0		c17	c16	c15	c14	c13	c12	c11	c10

c0-c17 0 Mode for channel c*xx* off

1 Mode for channel c*xx* on

0x1f80_1d88 Voice ON (0-15)

0x1f80_1d8a Voice ON (16-23)

Sets the current voice to key on. (ie. start ads)

0x1f80_1d8c Voice OFF (0-15)

0x1f80_1d8e Voice OFF (16-23)

Sets the current voice to key off.(ie. release)

0x1f80_1d90 Channel FM (pitch lfo) mode (0-15)

0x1f80_1d92 Channel FM (pitch lfo) mode (16-23)

Sets the channel frequency modulation. Uses the previous channelas modulator.

0x1f80_1d94 Channel Noise mode (0-15)

0x1f80_1d96 Channel Noise mode (16-23)

Sets the channel to noise.

0x1f80_1d98 Channel Reverb mode (0-15)

0x1f80_1d9a Channel Reverb mode (16-23)

Sets reverb for the channel. As soon as the sample ends, the reverb for that channel is turned off.

0x1f80_1d9c Channel ON/OFF (0-15)

0x1f80_1d9e Channel ON/OFF (16-23)

Returns wether the channel is mute or not.

0x1f80_1da2 Reverb work area start

15	0
MVol	

Revwa 0x0000-0xffff Reverb work area start in sound buffer /8

0x1f80_1da4 Sound buffer IRQ address.

15	0
IRQa	

IRQa 0x0000-0xffff IRQ address in sound buffer /8

0x1f80_1da6 Sound buffer IRQ address.

15	0
Sba	

Sba 0x0000-0xffff Address in sound buffer divided by eight. Next transfer to this address.

0x1f80_1da8 SPU data

15	0

Data forwarding reg, for non DMA transfer.

0x1f80_1daa SPU control sp0

15	14	13	8	7	6	5	4	3	2	1	0
En	Mu	Noise		Rv	Irq	DMA		Er	Cr	Ee	Ce

En 0 SPU off

1 SPU on

Mu 0 Mute SPU

1 Unmute SPU

Noise Noise clock frequency

Rv 0 Reverb Disabled

1 Reverb Enabled

Irq 0 Irq disabled

1 Irq enabled

DMA 00

01 Non DMA write (transfer through data reg)

10 DMA Write

11 DMA Read

Er 0 Reverb for external off

1 Reverb for external on

Cr 0 Reverb for CD off

1 Reverb for CD on

Ee 0 External audio off

1 External audio on

Ce 0 CD audio off

1 CD audio on

0x1f80_1dac SPU status

15	0

In SPU init routines this register get loaded with 0x4.

0x1f80_1dae SPU status

15	12	11	10	9	0
		Dh	Rd		

Dh 0 Decoding in first half of buffer

1 Decoding in second half of buffer

Rd 0 Spu ready to transfer

1 Spu not ready

Some of bits 9-0 are also ready/not ready states. More on that later. Functions that wait for the SPU to be ready, wait for bits a-0 to become 0.

0x1f80_1db0 CD volume left

0x1f80_1db2 CD volume right

15	14	0
P	CDvol	

CDvol 0x0000-0x7fff Set volume of CD input.

P 0 Normal phase.

1 Inverted phase.

0x1f80_1db4 Extern volume left

0x1f80_1db6 Extern volume right

15	14	0
P	Exvol	

Exvol 0x0000-0x7fff Set volume of External input.

P 0 Normal phase.

1 Inverted phase.

0x1dc0-&1dff Reverb configuration area

0x1f80_1dc0

0x1f80_1dc2

0x1f80_1dc4 Lowpass Filter Frequency. 7fff = max value= no filtering

0x1f80_1dc6 Effect volume 0 - 0x7fff, bit 15 = phase.

0x1f80_1dc8

0x1f80_1dca

0x1f80_1dcc

0x1f80_1dce Feedback

0x1f80_1dd0

0x1f80_1dd2

0x1f80_1dd4 Delaytime(see below)

0x1f80_1dd6 Delaytime(see below)

0x1f80_1dd8 Delaytime(see below)

0x1f80_1dda

0x1f80_1ddc

0x1f80_1dde

0x1f80_1de0 Delaytime(see below)

0x1f80_1de2

0x1f80_1de4

0x1f80_1de6

0x1f80_1de8

0x1f80_1dea

0x1f80_1dec

0x1f80_1dee

0x1f80_1df0

0x1f80_1df2

0x1f80_1df4 Delaytime

0x1f80_1df6 Delaytime

0x1f80_1df8

0x1f80_1dfa

0x1f80_1dfc

0x1f80_1dfe

Reverb

The SPU is equipped with an effect processor for reverb echo and delay type of effects. This effect processor can do one effect at a time, and for each voice you can specify wether it should have the effect applied or not.

The effect is setup by initializing the registers 0x1dc0 to 0x1ffe to the desired effect. I do not exactly know how these work, but you can use the presets below.

The effect processor needs a bit of sound buffer memory to perform it's calculations. The size of this depends on the effect type. For the presets the sizes are:

Reverb off 0x00000 Hall 0x0ade0

Room 0x026c0 Space echo 0x0f6c0

Studio small 0x01f40 Echo 0x18040

Studio medium 0x04840 Delay 0x18040

Studio large 0x06fe0 Half echo 03c00

The location at which the work area is location is set in register 0x1da2 and it's value is the location in the sound buffer divided by eight. Common values are as follows:

Reverb off 0xFFFE Hall 0xEA44

Room 0xFB28 Space echo 0xE128

Studio small FC18 Echo 0xCFF8

Studio medium 0xF6F8 Delay 0xCFF8

Studio large 0xF204 Half echo 0xF880

For the delay and echo effects (not space echo or half echo) you canspecify the delay time, and feedback. (range 0-127) Calculations are shownbelow.

When you setup up a new reverb effect, take the following steps:

-Turn off the reverb (bit 7 in sp0)

-Set Depth to 0

-First make delay & feedback calculations.

-Copy the preset to the effect registers

-Turn on the reverb

-Set Depth to desired value.

Also make sure there is the reverb work area is cleared, else you might get some unwanted noise.

To use the effect on a voice, simple turn on the corresponing bit in the channel reverb registers. Note that these get turned off autmatically when the sample for the channel ends.

Effect presets

copy these in order to 0x1dc0-0x1dfe

Reverb off:

```
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
```

Room:

0x007D, 0x005B, 0x6D80, 0x54B8, 0xBED0, 0x0000, 0x0000, 0xBA80

0x5800, 0x5300, 0x04D6, 0x0333, 0x03F0, 0x0227, 0x0374, 0x01EF

0x0334, 0x01B5, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000

0x0000, 0x0000, 0x01B4, 0x0136, 0x00B8, 0x005C, 0x8000, 0x8000

Studio Small:

0x0033, 0x0025 0x70F0 0x4FA8 0xBCE0 0x4410 0xC0F0 0x9C00

0x5280 0x4EC0 0x03E4 0x031B 0x03A4 0x02AF 0x0372 0x0266

0x031C 0x025D 0x025C 0x018E 0x022F 0x0135 0x01D2 0x00B7

0x018F 0x00B5 0x00B4 0x0080 0x004C 0x0026 0x8000 0x8000

Studio Medium:

0x00B1 0x007F 0x70F0 0x4FA8 0xBCE0 0x4510 0xBEF0 0xB4C0

0x5280 0x4EC0 0x0904 0x076B 0x0824 0x065F 0x07A2 0x0616

0x076C 0x05ED 0x05EC 0x042E 0x050F 0x0305 0x0462 0x02B7

0x042F 0x0265 0x0264 0x01B2 0x0100 0x0080 0x8000 0x8000

Studio Large:

0x00E3 0x00A9 0x6F60 0x4FA8 0xBCE0 0x4510 0xBEF0 0xA680

0x5680 0x52C0 0x0DFB 0x0B58 0x0D09 0x0A3C 0x0BD9 0x0973

0x0B59 0x08DA 0x08D9 0x05E9 0x07EC 0x04B0 0x06EF 0x03D2

0x05EA 0x031D 0x031C 0x0238 0x0154 0x00AA 0x8000 0x8000

Hall:

0x01A5 0x0139 0x6000 0x5000 0x4C00 0xB800 0xBC00 0xC000

0x6000 0x5C00 0x15BA 0x11BB 0x14C2 0x10BD 0x11BC 0x0DC1

0x11C0 0x0DC3 0x0DC0 0x09C1 0x0BC4 0x07C1 0x0A00 0x06CD

0x09C2 0x05C1 0x05C0 0x041A 0x0274 0x013A 0x8000 0x8000

Space Echo:

0x033D 0x0231 0x7E00 0x5000 0xB400 0xB000 0x4C00 0xB000

0x6000 0x5400 0x1ED6 0x1A31 0x1D14 0x183B 0x1BC2 0x16B2

0x1A32 0x15EF 0x15EE 0x1055 0x1334 0x0F2D 0x11F6 0x0C5D

0x1056 0x0AE1 0x0AE0 0x07A2 0x0464 0x0232 0x8000 0x8000

Echo:

0x0001 0x0001 0x7FFF 0x7FFF 0x0000 0x0000 0x0000 0x8100

0x0000 0x0000 0x1FFF 0x0FFF 0x1005 0x0005 0x0000 0x0000

0x1005 0x0005 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000

0x0000 0x0000 0x1004 0x1002 0x0004 0x0002 0x8000 0x8000

Delay:

0x0001 0x0001 0x7FFF 0x7FFF 0x0000 0x0000 0x0000 0x0000

0x0000 0x0000 0x1FFF 0x0FFF 0x1005 0x0005 0x0000 0x0000

0x1005 0x0005 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000

0x0000 0x0000 0x1004 0x1002 0x0004 0x0002 0x8000 0x8000

Half Echo:

0x0017 0x0013 0x70F0 0x4FA8 0xBCE0 0x4510 0xBEF0 0x8500

0x5F80 0x54C0 0x0371 0x02AF 0x02E5 0x01DF 0x02B0 0x01D7

0x0358 0x026A 0x01D6 0x011E 0x012D 0x00B1 0x011F 0x0059

0x01A0 0x00E3 0x0058 0x0040 0x0028 0x0014 0x8000 0x8000

Delay time calculation:

Choose delay time in range 0-0x7f. rXXXX means register 0x1f80_XXXX.

r1dd4 = dt*64.5 - r1dc0

r1dd6 = dt*32.5 - r1dc2

r1dd8 = r1dda + dt*32.5

r1de0 = r1de2 + dt*32.5

r1df4 = r1df8 + dt*32.5

r1df6 = r1dfa + dt*32.5

The CD-ROM

Overview

The PSX uses a special two speed CD-ROM that can stream at 352K/sec.It uses the following registers to control it

CDREG0 = 0x1f80_1800

CDREG1 = 0x1f80_1801

CDREG2 = 0x1f80_1802

CDREG3 = 0x1f80_1803

REGISTER FORMAT

CDREG0 write: 0 to send a command

1 to get the result

read: I/O status

bit 0 0 REG1 command send

1 REG1 data read

bit 1 0 data transfer finished

1 data transfer ready/in progress

bit 7 1 command being processed.

CDREG1 write: command

read: results

CDREG2 write: send arguments

write: 7 = flush arg buffer?

CDREG3 write: 7 = flush irq

read: hi nibble: unknown

low nibble: interrupt status

MODES FOR SETMODE

Mode	bit	function	
M_Speed	bit 7	0: normal speed	1: double speed
M_Strsnd	bit 6	0: ADPCM off	1: ADPCM on
M_Size	bit 5	0: 2048 byte	1: 2340 byte
M_Size2	bit 4	0:-	1: 2328 byte
M_SF	bit 3	bit 3 0: Channel off	1: Channel on
M_Report	bit 2	0: Report off	1: Report on
M_AutoPause	bit 1	0: AutoPause off 1	1: AutoPause on
M_CDDA	bit 0	0: CD-DA off	1: CD-DA on

These modes can be set using the setmode command,

Status bits:

Play	**bit 7**	**playing CD-DA**
Seek	**bit 6**	**seeking**
Read	**bit 5**	**reading data sectors**
ShellOpen	**bit 4**	**once shell open**
SeekError	**bit 3**	**seek error detected**
Standby	**bit 2**	**spindle motor rotating**
Error	**bit 1**	**command error detected**

These are the bit values for the status byte recieved from CD commands.

Interrupt values:

NoIntr	**0x00**	**No interrupt**
DataReady	**0x01**	**Data Ready**
Acknowledge	**0x02**	**Command Complete**

Complete	**0x03**	**Acknowledge**
DataEnd	**0x04**	**End of Data Detected**
DiskError	**0x05**	**Error Detected**

These are returned in the low nibble of CDREG3. First write a 1 to CDREG0 before reading CDREG3. When a command is completed it returns 3. To acknowledge an irq value after you've handled it, write a 1 to CDREG0 then a 7 to both CDREG2 and CDREG3. Another interrupt may be queued, so you should check CDREG3 again if 0 or if there's another interrupt to be handled.

Name	Command	Blocked	Paramater	Returns
Sync	**0x00**		**-**	**status**
Nop	**0x01**		**-**	**status**
Setloc	**0x02**		**min,sec,sector**	**status**
Play	**0x03**	**B**	**-**	**status**
Forward	**0x04**	**B**	**-**	**status**
Backward	**0x05**	**B**	**-**	**status**
ReadN	**0x06**	**B**	**-**	**status**
Standby	**0x07**	**B**	**-**	**status**
Stop	**0x08**	**B**	**-**	**status**
Pause	**0x09**	**B**	**-**	**status**
Init	**0x0a**		**-**	**status**

Mute	**0x0b**		**-**	**status**
Demute	**0x0c**		**-**	**status**
Setfilter	**0x0d**		**file,channel**	**status**
Setmode	**0x0e**		**mode**	**status**
Getparam	**0x0f**		**-**	**status,mode,file?,chan?,?,?**
GetlocL	**0x10**		**-**	**min,sec,sector,mode,file,channel**
GetlocP	**0x11**		**-**	**track,index,min,sec,frame,amin, asec,aframe**
GetTN	**0x13**		**-**	**status,first,total (BCD)**
GetTD	**0x14**		**rack(BCD)**	**status,min,sec (BCD)**
SeekL	**0x15**	**B**	*****	**status**
SeekP	**0x16**	**B**	*****	**status**
Test	**0x19**	**B**	**#**	**depends on parameter**
ID	**0x1A**	**B**	**-**	**success,flag1,flag2,00 4 letters of ID (SCEx)**
ReadS	**0x1B**	**B**	**-**	**status**
Reset	**0x1C**		**-**	**status**
ReadTOC	**0x1E**	**B**	**-**	**status**

*** These commands' targets are set using Setloc.**

 # Command 19 is really a portal to another set of commands.

B means blocking. These commands return an immediate result saying the command was started, but you need to wait for an IRQ in order to get real results.

Command descriptions:

Command Number	Command Name	Discription
0x00	Sync	Command does not succeed until all other commands complete. This can be used for synchronization - hence the name.
0x01	Nop	Does nothing; use this if you just want the status.
0x02	Setloc	This command, with its parameters, sets the target for commands with a * for their parameter list.
0x03	Play	Plays audio sectors from the last point seeked. This is almost identical to CdlReadS, believe it or not. The main difference is that this does not trigger a completed read IRQ. CdlPlay may be used on data sectors However, all sectors from data tracks are treated as 00, so no sound is played. As CdlPlay is reading, the audio data appears in the sector buffer, but is not reliable. Game Shark "enhancement CDs" for the 2.x and 3.x versions used this to get around the PSX copy protection.
0x04	Forward	Seek to next track ?
0x05	Backward	Seek to beginning of current track, or previous track if early in current track (like a CD player's back button)
0x06	ReadN	Read with retry. Each sector causes an IRQ (type 1) if ModeRept is on (I think). ReadN and ReadS cause errors if you're trying to read a non-PSX CD or audio CD without a mod chip.
0x07	Standby	CD-ROM aborts all reads and playing, but continues spinning. CD-ROM does not attempt to keep its place.
0x08	Stop	Stops motor. Official way to restart is 0A, but almost any

		command will restart it.
0x09	Pause	Like Standby, except the point is to maintain the current location within reasonable error.
0x0A	Init	Multiple effects at once. Setmode = 00, Standby, abort all commands.
0x0B	Mute	Turn off CDDA stream to SPU.
0x0C	Demute	Turn on CDDA streaming to SPU.
0x0D	Setfilter	Automatic ADPCM (CD-ROM XA) filter ignores sectors except those which have the same channel and file (parameters) in their subheader area. This is the mechanism used to select which of multiple songs in a single XA to play. Setfilter does not affect actual reading (sector reads still occur for all sectors).
0x0E	Setmode	Sets parameters such as read mode and spin speed. See chart above the command list.
0x0F	Getparam	returns status, mode, file, channel, ?, ?
0x10	GetlocL	Retrieves first 6 (8?) bytes of last read sector (header) This is used to know where the sector came from, but is generally pointless in 2340 byte read mode. All results are in BCD ($12 is considered track twelve, not eighteen) Command may execute concurrently with a read or play (GetlocL returns results immediately).
0x11	GetlocP	Retrieves 8 of 12 bytes of sub-Q data for the last-read sector. Same purpose as GetlocL, but more powerful, and works while playing audio. All results are in BCD. See note
0x13	GetTN	Get first track number and number of tracks in the TOC.
0x14	GetTD	Gets start of specified track (does it return sector??)
0x15	SeekL	Seek to Setloc's location in data mode (can only seek to data sectors, but is accurate to the sector)
0x16	SeekP	Seek to Setloc's location in audio mode (can seek to any sector, but is only accurate to the second)
0x19	Test	This function has many subcommands that are completely different. See ending notes

NOTES

- **the sub-Q fromat is as follows**

track: track number ($AA for lead-out area)

index: index number (INDEX lines in CUE sheets)

min: minute number within track

sec: second number within track

frame: sector number within "sec" (0 to 74)

amin: minute number on entire disk

asec: second number on entire disk

aframe: sector number within "asec" (0 to 74)

- **Test subcommands**

1A ID

Returns copy protection status. StatError for invalid data CD, StatStandby for valid PSX CD or audio CD. The following bits I'm unsure about, but I think the 3rd byte has $80 bit for "CD denied" and $10 bit for "import". $80 = copy, $90 = denied import, $10 = accepted import (Yaroze only). The 5th through 8th bytes are the SCEx ASCII string from the CD.

1B ReadS

Read without automatic retry.

1C Reset

Same as opening and closing the drive door.

1E ReadTOC

Reread the Table of Contents without reset.

To send a command:

- First send any arguments by writing 0 to CDREG0, then all arguments sequentially to CDREG2

- Then write 0 to CDREG0, and the command to CDREG1.

To wait for a command to complete:

- Wait until a CDrom irq occurs (bit 3 of the interrupt regs) The cause of the cdrom irq is in the low nibble of CDREG3. This is usually 3 on a succesful comletion. Failure to complete the command will result in a 5. If you don't wish to use irq's you can just check for the low nibble of cdreg3 to become something other than 0, but make sure it doesn't get cleared in any irq setup by the bios or some such.

To Get the results

- Write a 1 to CDREG0, then read CDREG0, If bit 5 is set, read a return value from CDREG1, then read CDREG0 again repeat until bit 5 goes low.

To Clear the irq

- After command completion the irq cause should be cleared, do this by writing a 1 to CDREG0 then 7 to CDREG2 and CDREG3. My guess is that the write to CDREG2 clears the arguments previously set from some buffer. Note that irq's are queued, and if you clear the current, another may come up directly..

To init the CD:

-Flush all irq's

-CDREG0=0

-CDREG3=0

-Com_Delay=4901 ($1f801020)

-Send 2 NOP's

-Command $0a, no args.

-Demute

To set up the cd for audio playback

CDREG0=2

CDREG2=$80

CDREG3=0

CDREG0=3

CDREG1=$80

CDREG2=0

CDREG3=$20

Also don't forget to init the SPU. (CDvol and CD enable especially)

You should not send some commands while the CD is seeking. (ie. status returns with bit 6 set.) Thing is that the status only gets updated after a new command. I haven't tested this for other command, but for the play command ($03) you can just keep repeating the command and checking the status returned by that, for bit 6 to go low(and bit 7 to go high in this case) If you don't and try to do a getloc directly after the play command reports it's done, the cd will stop. (I guess the cd can't get it's current location while it's seeking, so the logic stops the seek to get an exact fix, but never restarts..)

19 subcommands.

For one reason or another, there is a counter that counts the number of SCEx strings received by the CD-ROM controller.

Be aware that the results for these commands can exceed 8 bytes.

0x04	Read SCEx counter (returned in 1st byte?)
0x05	Reset SCEx counter. This also sets 1A's SCEx response to 00 00 00 00, but doesn't appear to force a protection failure.
0x20	Returns an ASCII string specifying where the CD-ROM firmware is intended to be used ("for Japan", "for U/C").
0x22	Returns a chip number inside the PSX in use.
0x23	Returns another chip number.
0x24	Returns yet another chip number. Same as 22's on some PSXs

Root Counters

Overview

Root counters are timers in the PSX. There are 4 root counters.

Counter	Base address	Synced to
0	0x1f80_1100	pixelclock
1	0x1f80_1110	horizontal retrace
2	0x1f80_1120	1/8 system clock
3		vertical retrace

Each have three registers, one with the current value, one with the counter mode, and one with a target value.

0x11*n*0 Count [read]

31	16	15	0
Garbage		Count	

Count Current count value, 0x0000-0xffff

Upper word seems to contain only garbage.

0x11*n*4 Mode [read/write]

31	10	9	8	7	6	5	4	3	2	1	0
Garbage		Div	Clc		Iq2		Iq1	Tar			En

En 0 Counter running

1 Counter stopped (only counter 2)

Tar 0 Count to $ffff

1 Count to value in target register

Iq1 Set both for IRQ on target reached.

Iq2

Clc 0 System clock (it seems)

1 Pixel clock (counter 0)

Horizontal retrace (counter 1)

Div 0 System clock (it seems)

1 1/8 * System clock (counter 2)

When Clc and Div of the counters are zero, they all run at the same speed. This speed seems to be about 8 times the normal speed of root counter 2, which is specified as 1/8 the system clock.

Bits 10 to 31 seem to contain only garbage.

0x11*n*8 Count [read/write]

31	16	15	0

Garbage	Target

Target Target value, 0x0000-0xffff

Upper word seems to contain only garbage.

Quick step-by-step:

To set up an interrupt using these counters you can do the following:

1 - Reset the counter. (Mode = 0)

2 - Set its target value, set mode.

3 - Enable corresponding bit in the interrupt mask register ($1f801074)

bit 3 = Counter 3 (Vblank)

bit 4 = Counter 0 (System clock)

bit 5 = Counter 1 (Hor retrace)

bit 6 = Counter 2 (Pixel)

4 - Open an event. (Openevent bios call - $b0, $08)

With following arguments:

a0-Rootcounter event descriptor or'd with the counter number.

($f2000000 - counter 0, $f2000001 - counter 1,$f2000002 - counter 2, $f2000003 - counter 3)

a1-Spec = $0002 - interrupt event.

a2-Mode = Interrupt handling ($1000)

a3-Pointer to your routine to be excuted.

The return value in V0 is the event identifier.

5 - Enable the event, with the corresponding bioscall ($b0,$0c) with the identifier as argument.

6 - Make sure interrupts are enabled. (Bit 0 and bit 10 of the COP0 status register must be set.)

Your handler just has to restore the registers it uses, and it should terminate with a normal jr ra.

To turn off the interrupt, first call disable event ($b0, $0d) and then close it using the Close event call ($b0,$09) both with the event number as argument.

Controllers

Overview

The PSX uses a 9 pin device connecter for use with the PSX controller. The controller port is exactly the same electricly as the memory card port. The only difference is the device driver that uses it, and it's external port shape. The controllers are accessed via the InitPAD StartPAD, StopPAD, PAD_init, and PAD_dr BIOS commands. These are covered in detail within the BIOS section of this document. The controller is a type of "smart device" that communicates data serially via the port. Port informaton is as follows.

Pin	signal	dir	active	description
1	dat	in	pos	data from pad or memory-card
2	cmd	out	pos	command data to pad or memory-card
3	+7V	--	--	+7.6V power source for CD-ROM drive
4	gnd	--	--	
5	+3V	--	--	+3.6V power source for system
6	sel	out	neg	select pad or memory-card
7	clk	out	--	data shift clock
8	--	--	--	N.A.
9	ack	in	neg	acknowladge signal from pad or memory-card

- 1) direction(in/out) is based from PSX
- 2) metal edge in pad connecter is connected pin 4 and sheald calbe.
- 3) signal SEL in PAD1, PAD2 is separated.

Comminucation timing chart

Timing is compatible in the PAD as well as the Memory-card.

Overview

```
SEL- |_____________________________________________________________|

CLK |||||||| |||||||| |||||||| |||||||| ||||||||

CMD X 01h XXXX 42h XXXX 00h XXXX 00h XXXX 00h XXXX
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

DAT -----XXXXXXXXXXXXX ID XXXX 5Ah XXXX key1 XXXX key2 XXXX-----
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

ACK- ---------------|_|---------|_|---------|_|---------|_|----------------
```

Top command. First comminucation(device check)

```
SEL- |__________________________________________________________________
```

```
                ______ _ _ _ _ _ _ _ __________________ _ _ _ _
              CLK |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_|
                                   __________ ___
        CMD |_________________________________________________| |_______
                                        ____
   DAT -----XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX |___________
 ACK- ----------------------------------------------|___|--------------------

                              X = none, - = Hi-Z
```

- **0x81 is memory-card, 0x01 is standard-pad at top command.**
- **serial data transfer is LSB-First format.**
- **data is down edged output, PSX is read at up edge in shift clock.**
- **PSX expects No-connection if not returned Acknoledge less than 100 usec.**
- **clock pluse is 250KHz.**
- **no need Acknoledge at last data.**
- **Acknoledge signal width is more than 2 usec.**
- **time is 16msec between SEL from previous SEL.**
- **SEL- for memory card in PAD access.**

Communication format with the Pad

After the command 0x01h is sent to the pad drom the system, the pad replies with a one-byte PAD ID(0x5A), then it will send a 2-byte key code and extended code.

Normal Pad timing flow ->

	10000000	1000010	1011010	1234567	1234567
CMD	01h	42h	00h	00h	00h

	xxxxxxxx	10000010	10100101	1234567	1234567
DAT	----	41h	5ah	SW.1	SW.2

data contents of normal PAD.(push low)

byte	b7	b6	b5	b4	b3	b2	b1	b0	
0	---								N.A.
1	0x41								'A'
2	0x5a								'Z'
3	LEFT	DOWN	RGHT	UP	STA	1	1	SEL	
4	Square	X	O	Triangle	R1	L1	R2	L2	

data contents NEGCON(NAMCO analog controler, push low)

byte	b7	b6	b5	b4	b3	b2	b1	b0	
0	-----								N.A.
1	0x23								
2	0x5a								'Z'
3	LEFT	DOWN	RGHT	UP	STA	1	1	1	
4	1	1	A	B	R	1	1	1	
5	handle data right:0x00, center:0x80								
6	I button ADC data (7bit unsigned)								00h to 7Fh
7	II button ADC data								00h to 7Fh
8	L button ADC data								00h to 7Fh

unknown data bit length in +6 to +8 ADC datas. (7 or 8 may be)

mouse data contents(push low)

byte	b7	b6	b5	b4	b3	b2	b1	b0	
0	-----								N.A.
1	0x12								

2	0x5a								'Z'
3	1	1	1	1	1	1	1	1	
4	1	1	1	1	LEFT	RGHT	0	0	
5	V moves 8bitSigned up:+,dwn:-,stay:00								
6	H moves 8bitSigned up:+,dwn:-,stay:00								

Memory cards

Memory Card Format

The memory card for the PSX is 128 kilobytes of non-volatile RAM. This is split into 16 blocks each containing 8 kilobytes each. The very first block is is a header block used as a directory and file allocation table leaving 15 blocks left over for data storage.

The data blocks contains the program data file name, block name, icon, and other critical information. The PSX accesses the data via a "frame" method. Each block is split into 64 frames, each 128 bytes. The first frame (frame 0) is the file name, frames 1 to 3 contain the icon, (each frame of animation taking up one frame) leaving the rest of the frames for save data.

Terms and Data Format

This is the format of the various objects within the memory card.

File Name

Country code(2 bytes)+Product number(10 bytes)+identifier(8 bytes) An example of a product number is SCPS-0000. The identifier is a variation on the name of the game, for example FF8 will be FF0800, FF0801. The format if the product is 4 characters, a hyphen, and then 5 characters. The actula characters don't really matter. With a PocketStation program, the product ID is a monochrome icon, a hyphen and the later part containing a "P"

Country Code

In Japan the code is BI, Europe is BE, and America is BA. An American PSX and use memory saves with the BI country code.

Title

The title is in Shift-JIS format with a max if 32 characters. ASCII can be used as ASCII is a subset of Shift-JIS.

XOR Code

This is a checksum. Each byte is XORed one by one and the result is stored. Complies with the checksum protocol.

Link

This is a sequence of 3 bytes to link blocks togeather to form one continuous data block.

Data Size

Total Memory	128KB = 13,1072 bytes= 0x20000bytes
1 Block	8KB = 8192 bytes = 0x2000 bytes
1 Frame	128 bytes = 0x80 bytes

Header Frame

+0x00	'M' (0x4D)
+0x01	'C' (0x43)
+0x02 - 0x7E	Unused (0x00)

+0x7F	XOR code (usually 0x

Directory Frame

+0x00	Availible bocks upper 4 bits A - Availible 5 - partially used F - Unusable Lowe 4 bits 0 - Unused 1 - There is no link, but one wil 2 - mid link block 3 - terminiting link bl F - unusable Example A0 - Open block 51 - In use, there will be a link in 52 - In use, this is in a link and wi 53 - In use, this is the last i FF - Unusable
+0x01 - 0x03	00 00 00 When it's reservered it's I
+0x04 - 0x07	Use byte 00 00 00 - Open block middle link blo Block * 0x2000 - No link, but v (00 20 00 - one blocks will

	(00 40 00 - two blocks will (00 E0 01 - 15 blocks will
+0x08-0x09	Link order Block 0- If the bock isn't in a link or if it's the last link
+0x0A-0x0B	Country Code (BI, BA,
+0x0C+0x15	Product Code (AAAA-0 Japan SLPS, SCPS (from America SLUS, SCUS (fro Europe SLES, SCES (fro
+0x16-0x1D	Identifier This Number is created unique to the current game played. Meanin subsequent save has the same identifier, but it a new game is started fr
+0x1E-0x7E	Unused
0x7F	XOR Code
THE FOLLOWING DATA REPEATS FOR THE NEXT 15 BLOCKS	

Block Structure

Frame 0 Title Frame	0x00	'S' (0x5
	0x01	'C' (0x4
	+0x02	Icon Displa 00...No i 11...Icon has 1 frame o 12...Icon has 13...Icon has
	+0x03	Block Numb

	0x04 - 0x43	Title This is the title in Shift-JIS format, it al
	0x44 - 0x5F	Reserved This is used for the
	0x60 - 0x7F	Icon 16 Color P
Frame 1 Frame 3 Icon Frame	0x00 - 0x7F	Icon Bit 1 Frame of animaton = If there is no Icon for this
Frame 4 Data Frame	+0x00 - 0x7F	Save D

Link Block

Frame 1	+0x00 - 0x7F	Save Data

Data Transmission

Data is trasmitted with exactly the same protocol as the Pad data is trasmitted/revived. The pin out are exactly the same as well, the houseing, however is a different shape.

Serial I/O

The PSX has a 8 pin serial adapter that uses a non-standars protocol for data transmission and receiving. The pin outs are pictured here.

The pot speed is able to go up to a maximum of 256K bps. Normally it's used at 56K. On connecton problems the port will attempt a reconnect, but may not fall back on a slower speed. The link cable is wired is such.

```
1 <-> 4

2 NC

3 <-> 6

4 <-> 1

5 <-> 8

6 <-> 3

7 <-> 7

8 <-> 5
```

The pins are like this (looking into the link cable connector looking into the pins of the cable connector) and the connector facing up) :

```
CABLE
______________________________
// \\
```

```
// \\
________________ ________________
/ UP \ / UP \
------------------ -------------------
LEFT |1 2 3 4 5 6 7 8 | RIGHT LEFT | 1 2 3 4 5 6 7 8 | RIGHT
------------------ -------------------
```

Parallel I/O

Overview

The Parallel prt is is a sort of a faux name. It's really an expantion port. Any device connected to this port will have access to everything on the local bus. The address that the PIO port resides on is from 0x1f00_0000-0x1f00_ffff The following is a pin diagram of the PIO.

Appendix A

Number systems

The Hexadecimal system is as follows

Decimal	Hexadecimal
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10

17	11
18	12
19	13
20	14
21	15
22	16
23	17
24	18
25	19
26	1A
27	1B
28	1C
29	1D
30	1E
31	1F
32	20
33	21
...	...
252	FC
253	FD
254	FE
255	FF

Appendix B

BIOS calls

1st column - Address to call

2nd column - Value of $t1 when calling

3rd column - Name of the function

Arguments whenever needed are passed by $a0,1,2,3 and at $sp+0x10 when more than 4 arguments.

0x00a0 - 0x0000 - int open(char *name , int mode)

0x00a0 - 0x0001 - int lseek(int fd , int offset , int whence)

0x00a0 - 0x0002 - int read(int fd , char *buf , int nbytes)

0x00a0 - 0x0003 - int write(int fd , char *buf , int nbytes)

0x00a0 - 0x0004 - close(int fd)

0x00a0 - 0x0005 - int ioctl(int fd , int cmd , int arg)

0x00a0 - 0x0006 - exit()

0x00a0 - 0x0007 - sys_b0_39()

0x00a0 - 0x0008 - char getc(int fd)

0x00a0 - 0x0009 - putc(char c , int fd)

0x00a0 - 0x000a - todigit

0x00a0 - 0x000b - double atof(char *s)

0x00a0 - 0x000c - long strtoul(char *s , char **ptr , int base)

0x00a0 - 0x000d - unsigned long strtol(char *s , char **ptr , int base)

0x00a0 - 0x000e - int abs(int val)

0x00a0 - 0x000f - long labs(long lval)

0x00a0 - 0x0010 - long atoi(char *s)

0x00a0 - 0x0011 - int atol(char *s)

0x00a0 - 0x0012 - atob

0x00a0 - 0x0013 - int setjmp(jmp_buf *ctx)

0x00a0 - 0x0014 - longjmp(jmp_buf *ctx , int value)

0x00a0 - 0x0015 - char *strcat(char *dst , char *src)

0x00a0 - 0x0016 - char *strncat(char *dst , char *src , int n)

0x00a0 - 0x0017 - int strcmp(char *dst , char *src)

0x00a0 - 0x0018 - int strncmp(char *dst , char *src , int n)

0x00a0 - 0x0019 - char *strcpy(char *dst , char *src)

0x00a0 - 0x001a - char *strncpy(char *dst , char *src , int n))

0x00a0 - 0x001b - int strlen(char *s)

0x00a0 - 0x001c - int index(char *s , int c)

0x00a0 - 0x001d - int rindex(char *s , int c)

0x00a0 - 0x001e - char *strchr(char *c , int c)

0x00a0 - 0x001f - char *strrchr(char *c , int c)

0x00a0 - 0x0020 - char *strpbrk(char *dst , *src)

0x00a0 - 0x0021 - int strspn(char *s , char *set)

0x00a0 - 0x0022 - int strcspn(char *s , char *set)

0x00a0 - 0x0023 - strtok(char *s , char *set)

0x00a0 - 0x0024 - strstr(char *s , char *set)

0x00a0 - 0x0025 - int toupper(int c)

0x00a0 - 0x0026 - int tolower(int c)

0x00a0 - 0x0027 - void bcopy(void *src , void *dst , int len)

0x00a0 - 0x0028 - void bzero(void *ptr , int len)

0x00a0 - 0x0029 - int bcmp(void *ptr1 , void *ptr2 , int len)

0x00a0 - 0x002a - memcpy(void *dst , void *src , int n)

0x00a0 - 0x002b - memset(void *dst , char c , int n)

0x00a0 - 0x002c - memmove(void *dst , void *src , int n)

0x00a0 - 0x002d - memcmp(void *dst , void *src , int n)

0x00a0 - 0x002e - memchr(void *s , int c , int n)

0x00a0 - 0x002f - int rand()

0x00a0 - 0x0030 - void srand(unsigned int seed)

0x00a0 - 0x0031 - void qsort(void *base , int nel , int width , int (*cmp)*void *,void *))

0x00a0 - 0x0032 - double strtod(char *s , char *endptr)

0x00a0 - 0x0033 - void *malloc(int size)

0x00a0 - 0x0034 - free(void *buf)

0x00a0 - 0x0035 - void *lsearch(void *key , void *base , int belp , int width , int (*cmp)(void * , void *))

0x00a0 - 0x0036 - void *bsearch(void *key , void *base , int nel , int size , int (*cmp)(void * , void *))

0x00a0 - 0x0037 - void *calloc(int size , int n)

0x00a0 - 0x0038 - void *realloc(void *buf , int n)

0x00a0 - 0x0039 - InitHeap(void *block , int n)

0x00a0 - 0x003a - _exit()

0x00a0 - 0x003b - char getchar(int fd)

0x00a0 - 0x003c - putchar(char c , int fd)

0x00a0 - 0x003d - char *gets(char *s)

0x00a0 - 0x003e - puts(char *s)

0x00a0 - 0x003f - printf(char *fmt , ...)

0x00a0 - 0x0041 - LoadTest(char *name , XF_HDR *header)

0x00a0 - 0x0042 - Load(char *name , XF_HDR *header)

0x00a0 - 0x0043 - Exec(struct EXEC *header , int argc , char **argc)

0x00a0 - 0x0044 - FlushCache()

0x00a0 - 0x0045 - void InstallInterruptHandler()

0x00a0 - 0x0046 - GPU_dw

0x00a0 - 0x0048 - int SetGPUStatus(int status)

0x00a0 - 0x0049 - GPU_cw

0x00a0 - 0x004a - GPU_cwb (not sure)

0x00a0 - 0x004d - int GetGPUStatus()

0x00a0 - 0x0049 - GPU_sync

0x00a0 - 0x0051 - LoadExec(char *name , int , int)

0x00x0 - 0x0052 - GetSysSp()

0x00a0 - 0x0054 - _96_init()

0x00a0 - 0x0055 - _bu_init()

0x00a0 - 0x0056 - _96_remove()

0x00a0 - 0x0057 - return 0 (it only does this)

0x00a0 - 0x0058 - return 0 (it only does this)

0x00a0 - 0x0059 - return 0 (it only does this)

0x00a0 - 0x005a - return 0 (it only does this)

0x00a0 - 0x005b - dev_tty_init

0x00a0 - 0x005c - dev_tty_open

0x00a0 - 0x005e - dev_tty_ioctl

0x00a0 - 0x005f - dev_cd_open

0x00a0 - 0x0060 - dev_cd_read

0x00a0 - 0x0061 - dev_cd_close

0x00a0 - 0x0062 - dev_cd_firstfile

0x00a0 - 0x0063 - dev_cd_nextfile

0x00a0 - 0x0064 - dev_cd_chdir

0x00a0 - 0x0065 - dev_card_open

0x00a0 - 0x0066 - dev_card_read

0x00a0 - 0x0067 - dev_card_write

0x00a0 - 0x0068 - dev_card_close

0x00a0 - 0x0069 - dev_card_firstfile

0x00a0 - 0x006a - dev_card_nextfile

0x00a0 - 0x006b - dev_card_erase

0x00a0 - 0x006c - dev_card_undelete

0x00a0 - 0x006d - dev_card_format

0x00a0 - 0x006e - dev_card_rename

0x00a0 - 0x0070 - _bu_init

0x00a0 - 0x0071 - _96_init

0x00a0 - 0x0072 - _96_remove

0x00a0 - 0x0078 - _96_CdSeekL

0x00a0 - 0x007c - _96_CdGetStatus

0x00a0 - 0x007e - _96_CdRead

0x00a0 - 0x0085 - _96_CdStop

0x00a0 - 0x0096 - AddCDROMDevice()

0x00a0 - 0x0097 - AddMemCardDevice()

0x00a0 - 0x0098 - DisableKernelIORedirection()

0x00a0 - 0x0099 - EnableKernelIORedirection()

0x00a0 - 0x009c - GetConf(int Event , int TCB , int Stack)

0x00a0 - 0x009d - GetConf(int *Event , int *TCB , int *Stack)

0x00a0 - 0x009f - SetMem(int size)

0x00a0 - 0x00a0 - _boot

0x00a0 - 0x00a1 - SystemError

0x00a0 - 0x00a2 - EnqueueCdIntr

0x00a0 - 0x00a3 - DequeueCdIntr

0x00a0 - 0x00a5 - ReadSector(count,sector,buffer)

0x00a0 - 0x00a6 - get_cd_status ??

0x00a0 - 0x00a7 - bufs_cb_0

0x00a0 - 0x00a8 - bufs_cb_1

0x00a0 - 0x00a9 - bufs_cb_2

0x00a0 - 0x00aa - bufs_cb_3

0x00a0 - 0x00ab - _card_info

0x00a0 - 0x00ac - _card_load

0x00a0 - 0x00ad - _card_auto

0x00a0 - 0x00ae - bufs_cb_4

0x00a0 - 0x00b2 - do_a_long_jmp()

0x00a0 - 0x00b4 - (sub_function)

0 - u_long GetKernelDate (date is in 0xYYYYMMDD BCD format)

1 - u_long GetKernel???? (returns 3 on cex1000 and cex3000)

2 - char *GetKernelRomVersion()

3 - ?

4 - ?

5 - u_long GetRamSize() (in bytes)

6 -> F - ??

0x00b0 - 0x0000 - SysMalloc (to malloc kernel memory)

0x00b0 - 0x0007 - DeliverEvent(class , event)

0x00b0 - 0x0008 - OpenEvent(class , spec , mode , func) (source code is corrected)

0x00b0 - 0x0009 - CloseEvent(event)

0x00b0 - 0x000a - WaitEvent(event)

0x00b0 - 0x000b - TestEvent(event)

0x00b0 - 0x000c - EnableEvent(event)

0x00b0 - 0x000d - DisableEvent(event)

0x00b0 - 0x000e - OpenTh

0x00b0 - 0x000f - CloseTh

0x00b0 - 0x0010 - ChangeTh

0x00b0 - 0x0012 - InitPad

0x00b0 - 0x0013 - StartPad

0x00b0 - 0x0014 - StopPAD

0x00b0 - 0x0015 - PAD_init

0x00b0 - 0x0016 - u_long PAD_dr()

0x00b0 - 0x0017 - ReturnFromException

0x00b0 - 0x0018 - ResetEntryInt

0x00b0 - 0x0019 - HookEntryInt

0x00b0 - 0x0020 - UnDeliverEvent(class , event)

0x00b0 - 0x0032 - int open(char *name,int access)

0x00b0 - 0x0033 - int lseek(int fd,long pos,int seektype)

0x00b0 - 0x0034 - int read(int fd,void *buf,int nbytes)

0x00b0 - 0x0035 - int write(int fd,void *buf,int nbytes)

0x00b0 - 0x0036 - close(int fd)

0x00b0 - 0x0037 - int ioctl(int fd , int cmd , int arg)

0x00b0 - 0x0038 - exit(int exitcode)

0x00b0 - 0x003a - char getc(int fd)

0x00b0 - 0x003b - putc(int fd,char ch)

0x00b0 - 0x003c - char getchar()

0x00b0 - 0x003d - putchar(char ch)

0x00b0 - 0x003e - char *gets(char *s)

0x00b0 - 0x003f - puts(char *s)

0x00b0 - 0x0040 - cd

0x00b0 - 0x0041 - format

0x00b0 - 0x0042 - firstfile

0x00b0 - 0x0043 - nextfile

0x00b0 - 0x0044 - rename

0x00b0 - 0x0045 - delete

0x00b0 - 0x0046 - undelete

0x00b0 - 0x0047 - AddDevice (used by AddXXXDevice)

0x00b0 - 0x0048 - RemoveDevice

0x00b0 - 0x0049 - PrintInstalledDevices

0x00b0 - 0x004a - InitCARD

0x00b0 - 0x004b - StartCARD

0x00b0 - 0x004c - StopCARD

0x00b0 - 0x004e - _card_write

0x00b0 - 0x004f - _card_read

0x00b0 - 0x0050 - _new_card

0x00b0 - 0x0051 - Krom2RawAdd

0x00b0 - 0x0054 - long _get_errno(void)

0x00b0 - 0x0055 - long _get_error(long fd)

0x00b0 - 0x0056 - GetC0Table

0x00b0 - 0x0057 - GetB0Table

0x00b0 - 0x0058 - _card_chan

0x00b0 - 0x005b - ChangeClearPad(int)

0x00b0 - 0x005c - _card_status

0x00b0 - 0x005d - _card_wait

0x00c0 - 0x0000 - InitRCnt

0x00c0 - 0x0001 - InitException

0x00c0 - 0x0002 - SysEnqIntRP(int index , long *queue)

0x00c0 - 0x0003 - SysDeqIntRP(int index , long *queue)

0x00c0 - 0x0004 - get_free_EvCB_slot()

0x00c0 - 0x0005 - get_free_TCB_slot()

0x00c0 - 0x0006 - ExceptionHandler

0x00c0 - 0x0007 - InstallExceptionHandlers

0x00c0 - 0x0008 - SysInitMemory

0x00c0 - 0x0009 - SysInitKMem

0x00c0 - 0x000a - ChangeClearRCnt

0x00c0 - 0x000b - SystemError ???

0x00c0 - 0x000c - InitDefInt

0x00c0 - 0x0012 - InstallDevices

0x00c0 - 0x0013 - FlushStdInOutPut

0x00c0 - 0x0014 - return 0

0x00c0 - 0x0015 - _cdevinput

0x00c0 - 0x0016 - _cdevscan

0x00c0 - 0x0017 - char _circgetc(struct device_buf *circ)

0x00c0 - 0x0018 - _circputc(char c , struct device_buf *circ)

0x00c0 - 0x0019 - ioabort(char *str)

0x00c0 - 0x001b - KernelRedirect(int flag)

0x00c0 - 0x001c - PatchA0Table

There are 3 more i know that arent called the same way as above:

MiPS R3000:

Exception() {

li $a0,0

syscall

}

EnterCriticalSection() {

li $a0,1

syscall

}

ExitCriticalSection() {

li $a0,2

syscall

}

Appendix C

GPU command listing

Overview of packet commands:

0x01 clear cache

0x02 frame buffer rectangle draw

0x20 monochrome 3 point polygon

0x24 textured 3 point polygon

0x28 monochrome 4 point polygon

0x2c textured 4 point polygon

0x30 gradated 3 point polygon

0x34 gradated textured 3 point polygon

0x38 gradated 4 point polygon

0x3c gradated textured 4 point polygon

0x40 monochrome line

0x48 monochrome polyline

0x50 gradated line

0x58 gradated line polyline

0x60 rectangle

0x64 sprite

0x68 dot

0x70 8*8 rectangle

0x74 8*8 sprite

0x78 16*16 rectangle

0x7c 16*16 sprite

0x80 move image in frame buffer

0xa0 send image to frame buffer

0xc0 copy image from frame buffer

0xe1 draw mode setting

0xe2 texture window setting

0xe3 set drawing area top left

0xe4 set drawing area bottom right

0xe5 drawing offset

0xe6 mask setting

Appendix D

Glossary of terms

PSX Playstation

SCEI Sony Computer Entertainment Incorporated (Sony of Japan)

SCEA Sony Computer Entertainment America (Sony of America)

SCEE Sony Computer Entertainment Europe (Sony of Europe)

GTE Geometry Transformation Engine

GPU Graphics Processing Unit

CPU Central Processing Unit

MDEC Motion DEcoding Chip

PIO Parallel Input/Output port

SPU Sound Processing Unit

BIOS Basic Input/Output System

Appendix E

Works cited – Bibliography

History of the Sony PlayStation taken from http://www.psxpower.com

The IDTR3051 ™ , R3052 ™ RISController ™ Hardware User's Manual Revision 1.4 July 15, 1994

©1992, 1994 Integrated Device Technology, Inc.

System.txt, cdinfo.txt, gpu.txt, spu.txt, gte.txt

doomed@c64.org http://psx.rules.org

gte-lite.txt

http://www.in-brb.de/~creature/

MDEC data from

jlo@ludd.luth.se and various people at PSXDEV mailing list

http://www.geocities.co.jp/Playtown/2004/

bero@geocities.co.jp

Memcard/PAD Data

HFB03536@nifty-serve.or.jp

PIO

bitmaster@bigfoot.com

Syscall

sgf22@cam.ac.uk

Mem card format: E-nash

http://www.vbug.or.jp/users/e-nash/

e-nash@i.am

Plus the many more at PSXDEV mailing list that helped ^_^

Exitcode 84905